FlexWAFE - an Architecture for Reconfigurable Image Processing Systems

FlexWAFE - eine Architektur für
rekonfigurierbare-Bildverarbeitungssysteme

Von der Fakultät für Elektrotechnik, Informationstechnik, Physik

der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung der Würde eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigte Dissertation

von:	Eng. Amilcar do Carmo Lucas
aus:	Almeirim, Portugal

eingereicht am:	19.04.2012
mündliche Prüfung am:	23.07.2012

Vorsitzender:	Prof. Dr.-Ing. Harald Michalik
1. Referent:	Prof. Dr.-Ing. Rolf Ernst
2. Referent:	Prof. Dr.-Ing. Holger Blume

2012

Kurzfassung

Kürzlich gab es eine Zunahme der Nachfrage nach hochauflösenden digitalen Medieninhalten in den Kino- und Fernsehenindustrien. Derzeit vorhandene Systeme entsprechen nicht den Anforderungen, oder sind zu teuer. Neue Hardware-Systeme und neuer Programmiertechniken sind erforderlich, um den hochauflösenden, hochwertigen, Bildanforderungen zu genügen und Kosten zu verringern. Die Industrie sucht eine flexible Architektur zur Ausführung mehrerer Anwendungen auf Standard-Komponenten, mit reduzierten Entwicklungszeiten.

Bis jetzt ist gängige Praxis, spezialisierten Architektur und Systeme zu entwickeln, die eine einzelne Anwendung zielen. Dieses hat wenig Flexibilität und führt zu hohe Entwicklungskosten, jede neue Anwendung ist fast von Grund auf neu konzipiert.

Unser Fokus war es, eine für Bild Verarbeitung geeignet Architektur zu entwickeln dass die Flexibilität hat mehrere Anwendungen an dieselbe FPGA-basierte Hardware-Plattform zu laufen. Die Neuheit in unserem Ansatz ist, dass wir Teile der Architektur zur Laufzeit rekonfigurieren, aber, ohne das Zeit und *constraints* strafe von FPGA Partielle-Rekonfiguration-Techniken. Die Architektur verwendet eine hierarchische Kontrollstruktur, die zur parallel Verarbeitung gut geeignet ist, und Single-Cycle-Latenz Rekonfiguration von Teilen der Verarbeitungs-Pipeline ermöglicht. Dieses wird unter Verwendung relativ weniger Ressourcen für die verteiltes Steuerung Strukturen erzielt.

Um das entwickelte Architektur zu testen ein komplexer Film-Korn-Rauschunterdrückung Algorithmus wurde auf einer von Thomson-Grass Valley entwickelt standard Hardware-Plattform umgesetzt. Das System erfüllt alle Anforderungen und hatte sehr wenig Last auf den hierarchischen Kontrollstrukturen, es gibt viel Wachstum Spielraum für viel kompliziertere Steuerunganforderungen.

Die Architektur ist zu anderen Hardwareplattformen portiert worden, und andere Anwendungen wurden ebenfalls implementiert. Der Laufzeitreconfigurability ist ein Schlüsselfaktor im Erfolg des FlexWAFE gewesen.

Abstract

Recently there has been an increase in demand for high-resolution digital media content in both cinema and television industries. Currently existing equipment does not meet the requirements, or is too costly. New hardware systems and new programming techniques are needed in order to meet the high-resolution, high-quality, image requirements and reduce costs. The industry seeks a flexible architecture capable of running multiple applications on top of standard off-the-shelf components, with reduced development time.

Until now, standard practice has been to develop specialized architectures and systems that target a single application. This has little flexibility and leads to high developments costs, every new application is designed almost from scratch.

Our focus was to develop an architecture that is suited to image stream processing and has the flexibility to run multiple applications using the same FPGA-based hardware platform. The novelty in our approach is that we reconfigure parts of the architecture at run-time, but without incurring in the time and added constraints penalty of FPGA-partial-reconfiguration techniques. The architecture uses a hierarchical control structure that is well suited to parallel processing, and allows single cycle latency reconfiguration of parts of the processing pipeline. This is achieved using relatively little resources for the distributed control structures.

To test the developed architecture a complex film-grain noise reduction algorithm was implemented on an off-the-shelf hardware platform developed by Thomson-Grass Valley. The system meet all the requirements and had very little load on the hierarchical control structures, there is growth headroom for much complexer control demands.

The architecture has been ported to other hardware platforms, and other applications have been implemented as well. The run-time reconfigurability has proven to be a key factor in the success of the FlexWAFE.

Acknowledgments

I would like to thank my thesis advisor Professor Rolf Ernst for the opportunity and trust he gave me when he offered me the job position after a short three month DAAD internship at his institute. I am very grateful for the motivation, guidance and insightful suggestions he gave me throughout my work. Furthermore I would like to thank Professor Harald Michalik for chairing the examination committee and Professor Holger Blume for the co-examination.

I would like to express my deepest gratitude to my college and friend Dr. Sven Heithecker who shared the office with me for many years and worked together with me on the same project. I've learned a lot from him, our cooperative brainstorming sessions helped shape the FlexWAFE architecture and he also helped me adapt to a new country. Many thanks go also to office college Henning Sahlbach, who supported me in the final stages of the thesis and built my Dr. hat[1]. I would like to thank Dr. Marek Jersak for the trust he gave me after the short internship, and for the friendship.

An important role in this thesis was also played by the excellent work environment provided by all the colleagues at the institute. I am thankful for the *Kaffee Runde* discussions, the institute excursions, the Christmas parties, and the extracurricular *hat building* opportunities provided by the institute.

On a more personal note I would like to thank my loving wife for all the support and encouragement she gave me throughout this thesis.

Most importantly of all, I would like to thank my parents, for the patience they had when educating me. And for the lovely way they found to bring-up technical curiosity in me. At age three I was already sure I wanted to be an electronics engineer. Thank you.

[1] PhD. candidates at IDA get a big, heavy, personalized microcontroller controlled hat with lots of flashing lights and moving parts

Contents

List of Figures

1. Introduction

Digital image processing is a mature research field that continues to advance due to the number of very important applications such as: medical imaging, broadcasting, multimedia systems, robotics, vision and others that uses it. This thesis adds to the extensive body of work done in the image processing field.

The FPGA-based architecture developed in the context of this thesis is targeted at computationally demanding image processing systems. It was developed in cooperation with several research companies and universities during one German BMBF and one European Union funded research projects: FlexFilm and MORPHEUS.

Our goal was to develop a hardware architecture for digital film and as such section 1.1 is an introduction into digital film processing, image gathering, post-production and delivery, with a focus on post-production. It will give a general overview about the commonly used effects, the technologies used to implement these effects with examples of commercially available systems. This section is concluded with an overview of current and future digital film formats which will be referenced throughout this thesis.

Section 1.2 (pages 4ff) presents the first research project, FlexFilm, its motivation, a novel image processing algorithm and the hardware board developed to run it. The contributions of Dr. Sven Heithecker, a colleague at our institute, on the topic of communication and memory scheduling are then described in (Section 1.2.6, pages 12ff). Section 1.2.7 (pages 12ff), is a short introduction to FlexWAFE, the developed architecture to efficiently implement the requirements of this project, and the contributions of this thesis to the project.

Appendix E (pages 121ff) presents other publications by the author in the context of this thesis.

This chapter is then concluded with an outline of this thesis in section 1.3 (pages 14ff).

1.1. Digital Film Processing

The technical side of cinema production can be decomposed into three steps: image gathering, post-production and delivery. These steps are increasingly using digital signal processing techniques in order to improve quality and decrease production costs. The traditional cellulose acetate film is increasingly being replaced with digital data and referred to as *digital film*. The next paragraphs explain how this new medium is used.

1.1.1. Image Gathering

Image gathering is the process of collecting images and sound. Until recent years this was done exclusively with "mechanical" film cameras and separate sound recorders. Nowadays the technology has evolved to a point where commercially viable, all digital cameras can produce images with a similar quality and resolution. Examples of these are the Viper FilmStream from Thomson Grass Valley [119], the RED one from Red digital cinema [98], or the DALSA Origin from Dalsa [31]. These cameras capture frames of resolutions up to 4520×2540 pixels and up to 16 bits per pixel and Red Green Blue (RGB) color component. These new cameras are being increasingly used due to the advantages of digital data: ease of duplication without quality loss and amenability to computer based image processing. The cinema industry today still manufactures and uses film based "mechanical" cinema cameras, but it is predictable that in the future digital cameras will completely replace them.

1.1.2. Post Production

1.1.2.1. Application

Image processing is done nowadays using digital data manipulation and therefore, if a film camera was used, the parts of the film that require further processing are converted into digital data using a film scanner such as the Spirit 4K from Thomson Grass Valley [118], the Cintel Millennium 4K [23] or the FilmLight Northlight 2 [45]. If a digital camera was used this first step is unnecessary. Currently, the typical resolution of each digital image frame is 2048 x 1536 pixels with a color quantization of 10 to 16 bit per RGB component, but all of the aforementioned film scanners can produce images with up to 4096×3112 pixels, also referred to as 4K resolution.

The digital data also referred to as Digital Intermediate (DI), is then manipulated to change the appearance of the images, for example by color correction, noise reduction, chroma keying, digital scenery or digital characters.

Color correction is applied for example to correct different colors due to shootings at different times with varying daylight, or to correct sensor differences of film scanners (called "primary color correction") or to evoke specific moods (e.g. Lord of the Rings (LOTR): blue colors in Lorien, golden colors in Rivendell [94]).

Croma keying is used to change the background of a scene, for example to place an actor in a non-existing location (e.g. LOTR: actors in front of Mount Doom or Minas Tirith [5]). This is done by filming actors against a background consisting of a single color or a relatively narrow range of colors and later replace the portions of the image which match this color by the alternate background images.

Digital scenery uses 3D models of a virtual world to create scenes which for example do not exist in reality (e.g. LOTR: Minas Tirith, Moria).

Digital characters means that a digital actor is created on a computer and then superimposed over the background of a real image (e.g. LOTR: the fantasy creatures). This is very often

used in conjunction with motion capturing, where movement and gestures of a real actor is used to control the digital character (e.g. LOTR: the Gollum character [5]).

Of course, all techniques can be combined (e.g. LOTR: the various battles [112]).

1.1.2.2. Techniques

The manipulations described above are usually referred as Computer Generated Images (CGI) and are done with a mix of dedicated hardware, accelerated software or pure software.

Dedicated hardware uses special, single purpose, highly specialized hardware in systems like Pandora Revolution [90], Thomson Grass Valley Scream grain reducer [115], Quantel Pablo [96] or digitalvision DVNR [35]. This type of hardware usually relies on Field Programmable Gate Arrays (FPGA), Application Specific Integrated Circuits (ASIC) or Digital Signal Processors (DSP) to do the required high performance computations and because it is built and programmed with a limited set of functions in mind it is not very flexible.

One other limiting factor of these systems is that they have high development costs due to the special hardware used and the highly skilled professionals required to develop it. New developments in recent years have reduced the complexity of programming these systems by offering a high level programming environment that increases the programmer's productivity by increasing the level of abstraction at which they work [109, 61, 84].

Systems based on **hardware accelerated software** use a host computer, typically a standard PC, Apple Macintosh MAC [7] or SGI [107], where parts of the image processing algorithms run in software. The other parts of the algorithms run on one or more extension boards with a dedicated hardware accelerator. This solution harvests the performance benefits of the dedicated hardware while maintaining the flexibility and scalability of software based applications. Their development cost is lower than pure hardware systems because only a part of the system requires highly skilled developers.

Examples of these systems are Vinci Resolve from DaVinci [32], based on Aspex LineDancer [9], the Thomson Grass Valley Bones Software [117], using a NVidia Graphics Processing Unit (GPU) [88] as accelerator for some of its image processing tasks, the Lustre system from Autodesk [10] or the Cell Accelerator Board from Mercury [80] which uses the Cell Broadband Engine processor [67] from Sony, Toshiba and IBM [108, 120, 60].

Software has the advantage of being easier to program that the two previously described platforms. It's also more flexible and can do more complex algorithms like raytrace rendering, but in order to achieve acceptable performance, many computers need to be used in parallel. There are many software only products like Shake or After Effects from Apple [7], Inferno from Autodesk [10] or Cinepaint [22] to name but a few.

1.1.3. Delivery

The final step of the film production is the conversion from digital data back to celluloid based film print for distribution purposes. This is achieved using a Film recorder like the

ARRILASER Speed Performance from Arri [8] (using lasers), the Celco fury [19] (using a Cathode Ray Tube (CRT)) or Definity from CCG [33] (using a Liquid Crystal Display (LCD)).

In a very small, but fast increasing number of cinemas, digital projectors are available and the conversion to celluloid based film print is unnecessary. It is foreseeable that in the future all cinema rooms will have exclusively digital projectors.

1.1.4. Resolutions

Table 1.1 on the facing page shows a selection of currently used video and digital film resolutions and required data rates[1]. For a more detailed introduction to video formats see [65]. As can be seen, digital film resolutions are considerably higher compared to standard or even high definition TV, not only regarding pixels, but also regarding color resolution (RGB versus subsampled Luminance, Blue-difference, Red-difference (YCbCr)) and color component depth. This results in higher data rates and higher computational demands.

1.2. The FlexFilm Project

1.2.1. Usage of FPGAs

The processing power required by the applications mentioned in section 1.1 on page 1 is beyond the scope of today's single standard processor or DSP systems. Dedicated specialized hardware such as ASICs could deliver the required performance, however the initial development cost of these devices is very high and they amortize only for high volume products. This makes ASICs economically not viable for digital film processing systems, since these are targeted to post processing in film studios and therefore only have a very small market volume - of a few 100 devices per year. Multi-processor systems or systems consisting of dedicated stream processors are either too complex to program, too expensive, do not provide the desired processing power or enough internal data rate.

FPGAs however approach the flexibility of programmable processors and the performance of dedicated hardware. They provide a set of relatively low level hardware elements which can be dynamically configured and connected to implement the desired functionality. Figure 1.1 on page 6 shows that in the last decade the logic capacity of Xilinx FPGAs [126] increased by a factor of ~12 ; FPGAs from Altera [6] provide a similar performance. Due to these huge advances in technology FPGAs have become powerful enough to implement complex System-on-Chips (SoC).

Apart from development, rapid prototyping systems and firmware updates, the FPGA configuration usually remains fixed as soon as the desired functionality is implemented. On the other side, the flexibility can be used to implement different applications on the same hardware. In this case the FPGA is reconfigured when a new application is selected. Exactly this is the basic idea of the FlexFilm project.

[1] In this thesis, the correct expression "data rate" is preferred over the mostly used word "bandwidth". See section A.3 (pages 109f) for details.

Name		Active Resolution	Color Model	bpp[2]	FPS[1]	MiB/ Image[3]	Mpixels/ Second[3]	MB/ Second[3]
SDTV (BT.799)	PAL	720 × 576i[4]	YCbCrK 4:4:4:4	30	25	1.483	10.37	38.88
	NTSC	720 × 480i[5]	YCbCrK 4:4:4:4	30	29.97	1.236	10.37	38.88
HDTV (BT.1120)		1920 × 1080i[6]	YCbCrK 4:2:2:4	20	25	4.944	51.84	129.6
2K		2048 × 1536	RGB 4:4:4	30	24	11.25	75.50	283.1
2K × 2K[7]		2048 × 2048	RGB 4:4:4	30	24	15.00	100.7	377.5
4K		4096 × 3072	RGB 4:4:4	30	24	45.00	302.0	1,132
8K		8192 × 6144	RGB 4:4:4	30	24	180.0	1,208	4,530
				48	24	288.0	1,208	7,248

[1] full frames per second
[2] bits per pixel, without keying value
[3] active resolution, full frames, without keying value
[4] interlaced, total resolution 864 × 625i
[5] interlaced, total resolution 858 × 525i
[6] interlaced, total resolution 2376 × 1250i
[7] anamorph

1 MiB = 1024 KiB = 1024 Byte, see section A.1 (page 109) for details on binary prefixes

Table 1.1.: Digital Video and Film Formats

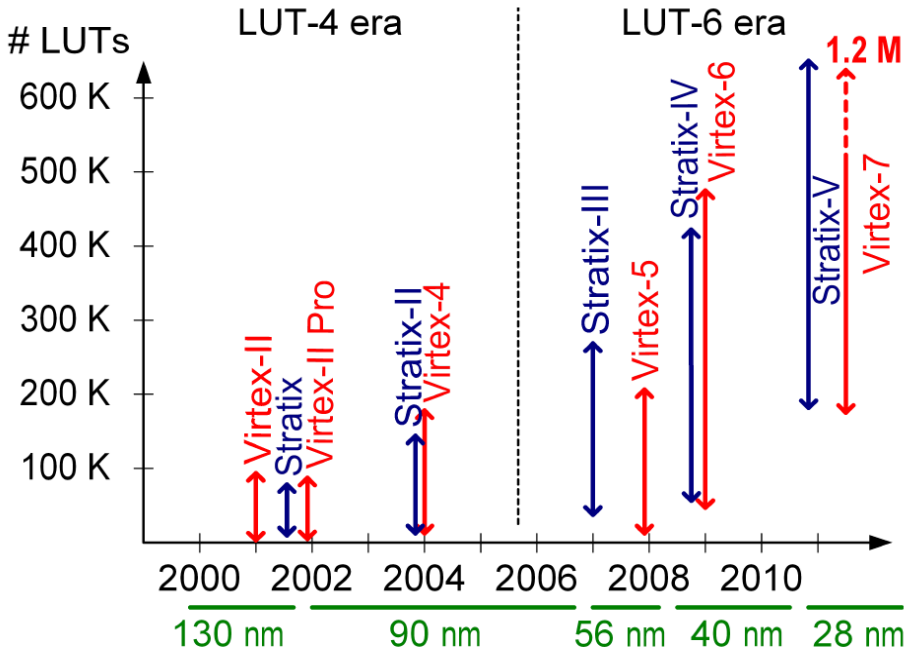


Figure 1.1.: Ten years of FPGA evolution: the increase in logic density over time and over the corresponding process technology.

1.2.2. Motivation

The approach of using the configurability of FPGAs to implement different kinds of applications was already successfully used in digital film processing systems [55, 56]. A closer look at the current design practice however revealed that the systems were designed with certain applications in mind, resulting in several drawbacks:

- At board level design specialized board-to-board and inter-FPGA communication channels as well as structures like additional filters and off-chip delay channels were used; the used FPGAs more or less matched the needs of the selected application. This prevented or complicated the reuse of these hardware and FPGA modules for other applications or as part of a scalable processing platform.
- The FPGAs were programmed by directly converting the specific algorithm into synthezisable Very High Speed Integrated Circuit HDL (VHDL) code, including Synchronous Dynamic Random Access Memory (SDRAM) controllers and intra-FPGA communication structures tailored to the algorithm's specific needs. The monolithic code was synthesized and implemented into FPGAs by use of standard tools like Xilinx ISE. This approach required not only knowledge of the algorithm to be implemented, but also in-depth understanding of chip-design practices and the underlying hardware architecture.

This approach leads to fully customized hardware that is difficult to reuse or adapt to changes and is less suitable for complex algorithms that require hardware scheduling.

Based on this insight, the three-year FlexFilm project was launched in January 2003, partially founded by the German Bundesministerium für Bildung und Foschung (BMBF). The goal of the project was to develop a digital film processing platform targeted to post processing in film studios with the following key features:

- *Platform based.* The system should not be designed for a single application, instead it should be a flexible, reusable platform for multiple current and future applications and algorithms. These applications include primary color correction, noise and grain reduction, scaling, format conversion, editing, encoding and digital effects such as blue boxing.
- *Real-time processing with smooth downgrade possibility.* The platform should be powerful enough to process a variety of applications in real-time at 2K resolution. If this is not possible, for example at higher resolutions (4K or even 8K), with highly complex applications (e.g. H264/AVC encoding) or because of currently unavailable hardware resources, a smooth degradation to near or non real-time processing should be possible.
- *Scalability.* The system should consist of several modules which should be combinable to extend the processing power if required. This required not only the availability of multiple flexible programmable processing units, but also demanded for a highly scalable performant communication structure. Common bus-like communication channels such as Peripheral Component Interconnect (PCI) or PCI-Extended (PCI-X) however do not scale well because of their shared nature and therefore are not an option. Instead, in the FlexFilm project, the idea was to use a flexible switched network like PCI-Express (PCIe).
- *Standard components.* Unlike previous digital film processing systems which were used either within a controlled and proprietary system environment or as a stand alone system (for example, "Scream Grain Reducer" together with "Spirit" film scanner series or the "LUTher Color Space Converter", see [114]), one goal was to design a system to be used in a non-specialized, preferably PC-based, system environment.

  Furthermore, as explained above ASICs are economically not viable for digital film post-processing systems. This is also true for other special components like video memory or dedicated video communication devices which are expensive due to their low market volume. Besides that, the availability of these components is not guaranteed - the possibility that highly specialized products quickly become discontinued due to product market in-acceptance should not be neglected. This might result in costly redesigns which is to be avoided.

  Therefore, in the FlexFilm project, the usage of standard components was proposed. This includes usage of PC technologies such as Synchronous Dynamic Random Access Memory (SDRAM) and communication frameworks systems like PCIe as well as other widely used devices like standard FPGAs and processors. Besides lower total cost there is a high chance that these components will not become discontinued quickly and therefore redesigns will be avoided or kept at a minimum.
- *Fast application development.* As explained above, the established development methodology with its long development times and requirements of hardware and chip

design knowledge is less suited for the new high complex digital film processing systems and algorithms. In order to simplify this process and to cut down time-to-market, development should focus more on the algorithm itself rather than on it's implementation.

Today, programming tools exist which simplify hardware design by converting a graphical representation (block diagram) of an algorithm into DSP-software or synthesisable VHDL code, for example Matlab/Simulink from MathWorks [113] (based on a time-discrete model), Ptolemy from Agilent [4] (based on SDF, see section 3.2), AutoESL from Xilinx [14] or the Mapping Tool from Compaan [25]. Using one of those tools together with a big modern Virtex 6 Chip the algorithm implementation the we will later introduce in this thesis would probably be possible today, but at the time of the FlexFilm project that was not possible because the FPGAs were smaller. Therefore, a technique was developed which combines programmability for reuse of components with the efficiency of dedicated hardware acceleration. The result which is described in this thesis is a very compact reconfigurable hardware that efficiently uses FPGA resources. Today, the architecture developed for FlexFilm can be used for:

- use the available resources efficiently; allowing to use smaller, cheaper FPGAs.
- map even bigger and complexer applications, that today do not fit in existing FPGAs.

1.2.3. Project partners and their work-packages

As explained, FlexFilm was a larger research project with numerous contributors. A concise overview shall explain the roles of the different partners and the following sections will detail them.

The Department of Electronic Circuits and Systems, working group Digital Image and Video Processing "digitale Bild und Videoverarbeitung" (pvk), at TU Ilmenau [34]

- developed the film grain noise reduction algorithm using Matlab explained in Section 1.2.4,
- helped in converting the algorithm from floating point to fixed point computation including word size determination,
- programmed a software-only version of the algorithm that was later used as reference for performance comparisons.

Thomson Grass Valley [116] at Weiterstadt was the project leader and was responsible for

- defining the market requirements like the example application, film format(s), board size and board type (PC-based extension card, no proprietary form factor),
- board development (schematics, layout as described in Section 1.2.5), manufacturing and initial test,
- major parts of the I/O-FPGA firmware, namely the integration of the Xilinx PCI-Express IP core, and the initial version of the PC software driver.

The Institute of Computer and Network Engineering (IDA) at TU Braunschweig [63] was responsible for

- system architecture and additional board development in cooperation with Grass Valley Germany (some of the decisions taken in Section 1.2.5);
- FPGA programming in the form of script and Very High Speed Integrated Circuit HDL (VHDL) code for:
  - the communication layer including Synchronous Dynamic Random Access Memory (SDRAM) access (Ph.D. thesis of Dr. Sven Heithecker [49]) described in Section 1.2.6 and
  - reusable, parameterizable modules to create stream processing data paths (this thesis) introduced in Section 1.2.7.

1.2.4. Example Application

One major goal was to create a platform capable of supporting a variety of applications. As a reference application, the FlexFilm project selected a complex algorithm to remove film grain noise from digitalized motion-picture film content while minimizing the negative impact on the images.

1.2.4.1. Film Grain Noise

Most motion-picture film material consists of an emulsion of silver halide crystals in gelatin (color: three layers for each primary color red, green, blue). When exposed to light the silver halide crystals get transformed into metallic silver grains. The light sensitivity of the film depends on the size of the silver halide crystals, the bigger their size, the more photo-sensitive the film is. These grains can be made small, if the film is to be used in highly illuminated scenarios, but on normally lit scenes a minimum granularity is required to correctly capture the motion images. During film development, the silver grains persist (black and write) or are replaced by dyes which form clusters centered around the original silver crystals (color). Although the eye is not able to see individual particles or dye clouds, graininess becomes visible especially for light-sensitive films (with large grains), large screens, short viewing distances or image magnifications. For moving pictures, the random placement of the grain on the film material causes the typical sensation of film grain noise.

While a certain noise level is usually desired for artistic reasons, a large noise level deteriorates visual image quality. Furthermore, a large noise level negatively affects video compression effectiveness. It is therefore desired to minimize the grain noise while keeping image details. For more information on film grain see [43, 44, 42] and the references therein.

1.2.4.2. The Algorithm

The implemented algorithm was specifically designed by Eichner et al. [43, 42] at TU-Ilmenau for removing film grain noise from digitalized motion-picture film content, while keeping as much image details as possible.

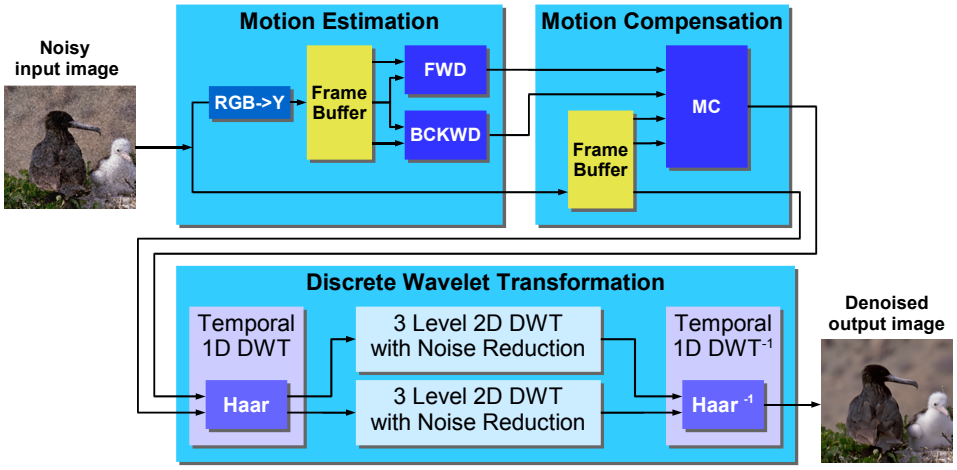


Figure 1.2.: Noise and grain reduction algorithm

Two consecutive frames from the same scene have mostly the same image content because they were shot within a small time interval, usually one-twenty-fourth of a second, between them and only objects that moved within that time frame are responsible for image content changes. In image areas where the content did not change, the film grain *level* remains the same, in the other areas the film grain level might change due to the dependency that it has on the light. On the other side, there is little temporal correlation between film grain *structure* on different frames because they are recorded on different locations on the film roll. This means that two consecutive frames from the same scene have mostly the same film grain level (due to content similarities) but a very different film grain structure (due to different locations on the film material).

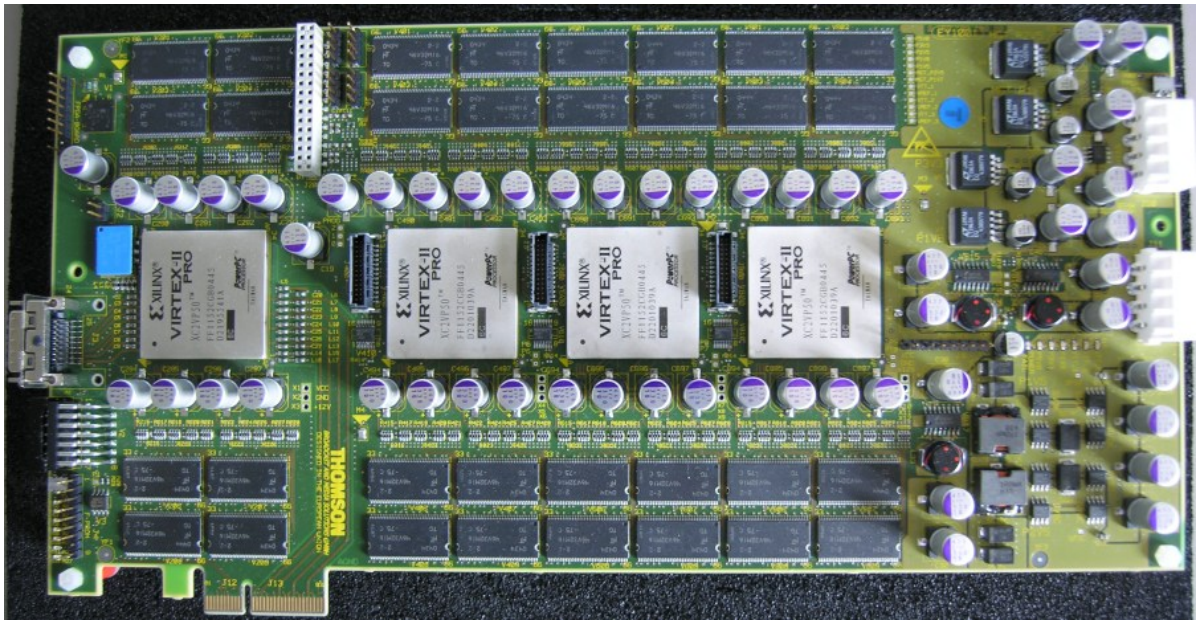
The algorithm implementation depicted in Figure 1.2 uses this fact to identify and remove the film grain noise. It basically subtracts two consecutive frames in such a way that the result is mostly film grain noise and eliminates the noise by filters.

Before subtracting the two frames, a motion compensation is applied to the second image to reduce artifacts generated by movement. To further improve the results, the subtraction operation is selectively performed between the current frame and areas from both the previous and the following frame, depending on which one has the most similar image content. It requires around $200x10^9$ operations per second to operate, and a software only implementation done by Eichner requires 1.2 minutes to process a single 2K image frame on a Intel Pentium4 2.4GHz, 1.5 GiB RAM (10.5 SpecInt2000-rate) under Microsoft WindowsXP.

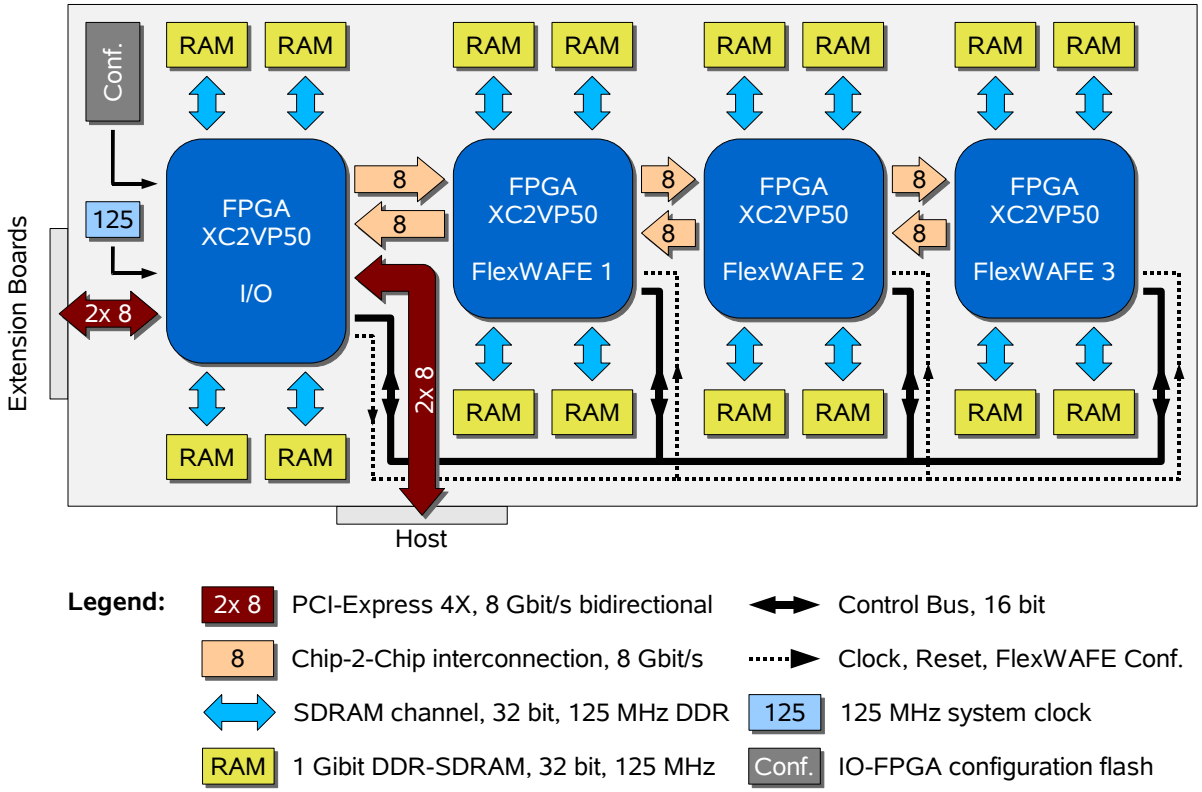
The implementation of this algorithm using the FlexWAFE architecture was used as a case-study in the FlexFilm project. It is a complex implementation of a complex algorithm and as such most of Chapter 6 on page 65 is dedicated to it.

1.2.5. Hardware Architecture

Based on the goals listed in Section 1.2.2 and the reference application described in the previous Section 1.2.4, the FlexFilm board was developed by Thomson Grass Valley. Figure 1.3



(a) Board



(b) Simplified block diagram

Figure 1.3.: FlexFilm EY1001 Processing Unit

shows a FlexFilm processing unit implemented as an PCI-Express extension card with 4 Xilinx XC2VP50-6 FPGAs.

1.2.5.1. FPGAs

Three of the FPGAs are used for image processing (**Flex**ible **W**eakly-programmable **A**dvanced **F**ilm **E**ngine (FlexWAFE) FPGAs), the 4th FPGA (I/O-FPGA) is used as bridge between the FlexWAFE FPGAs and the host PC. All FPGAs are synchronously clocked at 125 MHz.

Each FlexWAFE FPGA contains a large amount of logic resources to implement the *data paths* which are responsible for the stream or image processing [128]. Besides that, each FPGA contains two PowerPC processors for control and low computational tasks such as parameter calculation or heavily data-dependent tasks such as bit-stream encoding during image compression.

1.2.6. Communication Channels

For complex systems like the FlexFilm system, not only the processing itself, but also the communication between these elements, diverse FPGAs, boards and external memories are great challenges. The implementation of the communication infrastructure of the FlexFilm board was the focus of my colleague Dr. Sven Heithecker's thesis [49] and is illustrated in Figure 1.4 on the next page.

- *Inter-board* or *board-to-board communication*: for connecting multiple boards and the external storage system (film source and destination) a packet-based switched network is used. A small introduction to the implementation is given in section 3.6, but for in-depth explanation refer to chapter 3 of [49];
- *Inter-FPGA* or *chip-to-chip communication*: these fast, low-latency unidirectional channels connect data paths split across multiple FPGAs. An overview of the implementation is given in section 3.6, but for in depth explanation refer to chapter 4 of [49];
- *External RAM communication*: this channel covers the access of external (DDR-)SDRAM memory from the data paths and embedded CPUs. An overview of the requirements and the architecture is given in section 3.5, but for in depth explanation refer to chapter 5 of [49];
- *Data path communication*: the connections between the building blocks of the FlexWAFE IP core library which form the image processing data path where developed in the context of this thesis. Therefore an introduction will be given in Section 1.2.7 and the entire chapter 3 of this thesis is dedicated to it.

1.2.7. Contribution to FlexFilm

A set of image processing blocks capable of satisfying the requirements set in Section 1.2.2 and targeted at implementing the algorithm presented in Section 1.2.4 were developed. An

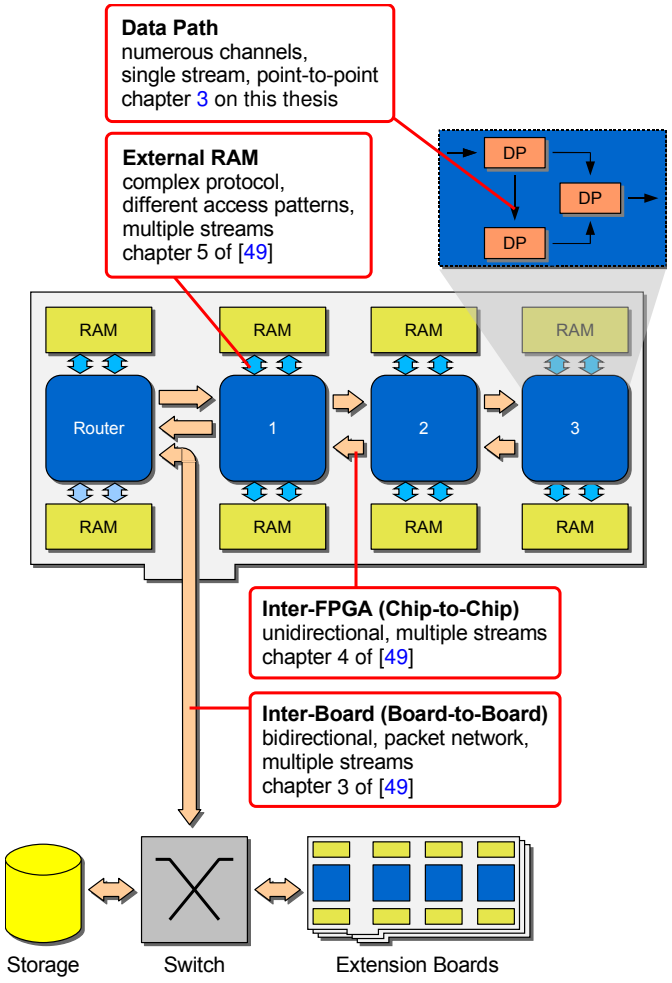


Figure 1.4.: FlexFilm communication channels

important goal was re-usability of blocks for other image processing applications and as such they were developed to be very flexible by using weakly-programmable structures (explained in detail in section 3.1). And so the **Flex**ible **W**eakly-programmable **A**dvanced **F**ilm **E**ngine (FlexWAFE) architecture was created, it will be further detailed in chapter 3 on page 25.

During the course of the FlexFilm project several hardware platforms were used to test the FlexWAFE blocks, but the final test of the full working algorithm was done on the FlexFilm board. The project objectives were met and even exceeded, the hardware implementation processed 2K image streams at 26 frames per second. At that time a software-only implementation in hand-written, optimized C code required around 1.2 minutes for a single 2K image frame on a Intel Pentium4 2.4GHz, 1.5 GiB RAM (10.5 SpecInt2000-rate) under Microsoft WindowsXP. The implementation of the film grain noise reduction algorithm using the FlexWAFE architecture was used as a case-study in the FlexFilm project and is described in chapter 6 on page 65.

The work covered by this thesis contributed to digital image processing in general and to the FlexFilm project in particular as follows:

- A weakly programmable, massively parallel configurable architecture for streaming applications on large data sets. It is targeted to FPGA based systems with general purpose logic blocks and small local memories;
- Development of processing elements for this architecture, consisting of a library parameterized data paths and local memory controllers and flexible address generators as well as their interfaces;
- Development of a distributed programming model for this architecture combining local processing element control and global algorithm control;
- Analysis of multiple implementation possibilities of the image processing algorithms targeted in the project. The main algorithm was decomposed in sub-algorithms, and multiple implementations for each one were analyzed. The implementation that best matched the requirements was then chosen and fully implemented and tested in hardware.

1.3. Thesis Outline

This thesis is structured as follows:

Chapter 2 presents related work and introduces many of the concepts used in this work.

Chapter 3 (pages 25ff) gives a more detailed presentation of the architecture in a bottom-up manner. It gives some insight about the existing low-level functionality and can be used as a starting point for anyone who wants to expand it.

In chapter 5 (pages 55ff), a top-down description of the programming model is presented as well as much of the reasoning behind the decisions of this work.

In chapter 6 (pages 65ff), we present an example application of FlexWAFE on the FlexFilm platform. It is the complex noise reduction filter introduced in this chapter. Chapter 6 details the hardware and software implementation as well as results.

Finally, in chapter 7 (pages 99ff), we summarize and conclude this thesis. Furthermore we give an outlook on the future of FlexWAFE as it is currently been used in the Flexelerator project in cooperation with Volkswagen and in the Digital Chameleon project in cooperation with the Institut für Computergraphik.

In Appendix A, pages 109ff, we summarize the terms used in this thesis.

Appendix B on page 111 explains the concept of memory based circular buffers. This is nothing new, but it will be presented here for the sake of completeness.

Appendix C on page 113 presents modifications and extensions to traditional asynchronous FIFOs implementations in order to improve their performance integration with the FlexFilm architecture.

Appendix D on page 115, explains the dynamic range of convolution-based signal processing operations. It should help plan the intermediate bit-widths of digital filters.

And finally Appendix E on page 121 summarizes other related publications by the author.

2. Processing platforms

This chapter will present some of the state-of-the-art platforms for digital image processing and streaming applications in section 2.1. Following that, section 2.2 (pages 21ff) describes some of the programing methodologies for FPGAs. Finally section 2.3 (pages 23ff) presents a comparison of the presented platforms and summarizes and concludes this chapter.

2.1. Processing platforms for digital film processing

This section starts by presenting platforms that were exclusively designed for image processing in digital film applications. Followed by platforms that are stream and/or vector oriented and were designed for a broader field of applications. And towards the end of this section, general purpose platforms are presented that are not very efficient at image processing.

2.1.1. Line Dancer

The *da Vinci Resolve* [32] product line is a system designed exclusively for image processing. It uses Power Plant accelerator boards based on the Aspex LineDancer processor [12, 66] that integrates one 32-bit Reduced Instruction Set Computer (RISC) processor, a DMA engine, 2 x 128KiB[1] on-chip memories, external memory interfaces, inter-chip connect and 4096 small processing elements, all operating in parallel, to provide an ultra-high performance processor that is fully software programmable. The array of small processors, each containing an Arithmetic Logic Unit (ALU), memory, and a high speed inter-processor communications network is known as ASProCore. All the processors work in parallel, performing instructions on data held in their local memory in a Single Instruction, Multiple Data (SIMD) configuration, allowing the array to perform up to 800 billion instructions per second. An ultra-high speed communications network allows processing elements to share data, with on-chip data rates up to 3,200 gigabits per second. Linedancer is programmed in C, which executes on the standard 32-bit RISC Central Processing Unit (CPU) core and the ASProCore array acts as a vector co-processor. This simple, uni-processor programming model makes Linedancer much easier to program than traditional parallel processing architectures. Integrated neighbor ports allow Linedancer chips to share data with adjacent chips, and for ASProCore Processing Elements to access data stored in other chips. Multiple devices can be cascaded together in a simple one-dimensional string to achieve more performance. This operation is transparent to the programmer, and no software changes are required to take advantage of the additional processors.

[1] 1Kib=1024 bits, see section A.1 (page 109) for details

2.1.2. GPU

The Thomson Bones system is based on an IBM PC compatible host and uses a NVidia GPU as accelerator for some of its image processing tasks. The Lustre system from Autodesk [10] also uses a GPU to accelerate it's processing tasks [77]. In recent years the power of the GPUs has increased to a point where it is feasible to use them as general purpose processors [46]. They are now capable coprocessors, and their high speed makes them useful for a variety of applications. An example of such an application is presented in [71], where GPUs are used to accelerate Matlab's image processing toolbox. All five academy award nominees for best visual effects of the year 2011 used GPUs [95]. The typical power efficiency of such processing systems is currently 50MFlOps/Watt [99].

Nvidia's Fermi [2] GPU architecture is used in the Tianhe-1A [59] supercomputer. This computer has set a new performance record of 2.507 petaflops, as measured by the LINPACK benchmark, making it the fastest system in the world as of November 2010 [81]. Tianhe-1A epitomizes modern heterogeneous computing by coupling massively parallel GPUs with multi-core CPUs, enabling significant achievements in performance, size and power. The system uses 7,168 NVIDIA® Tesla™ M2050 GPUs and 14,336 CPUs; it would require more than 50,000 CPUs and twice as much floor space to deliver the same performance using CPUs alone. The first Fermi based GPU, implemented with 3.0 billion transistors, has 16 stream multiprocessor (SM). Each SM features 32 CUDA (Compute Unified Device Architecture) processors. Each CUDA processor has a fully pipelined integer ALU and floating point unit. CUDA is the hardware and software architecture that enables NVIDIA GPUs to execute programs written with C, C++, Fortran, OpenCL, DirectCompute, and other languages. A CUDA program calls parallel kernels. A kernel executes in parallel across a set of parallel threads. One of the most important technologies of the Fermi architecture is its two-level, distributed thread scheduler. At the chip level, a global work distribution engine schedules thread blocks to various SMs, while at the SM level, each warp scheduler distributes warps of 32 threads to its execution units.

2.1.3. Cell Processor

The Cell Accelerator Board from Mercury [80] is a PCI Express accelerator card based on the Cell Broadband Engine processor from Sony Toshiba and IBM [68]. The Cell Accelerator Board solution offers the advantages of the Cell BE processor in a package designed for high-performance environments. The board provides approximately 180 GFLOPS for image processing, and graphics rendering of massive datasets. The Cell BE processor architecture is essentially a multicomputer-on-a-chip. It includes three main functional components:

- The Power processing element (PPE) has dual hardware multi-threading and a standard VMX vector processing engine. It has separate 32 KiB L1 data and instruction caches and 512 KiB of L2 cache. The processing power of the PPE is in addition to the 180 GFLOPS from the SPE array.
- In the array of eight synergistic processing elements (SPEs), each has a dual-issue pipeline, a 128-bit-wide vector processing engine, a very large register set (128 reg-

isters, each 128 bits wide), and 256 KiB of local store. Each SPE accesses system memory via its memory flow controller, which is a high-performance DMA engine.

- A high-speed data ring called the element interconnect bus (EIB) consists of two pairs of counter-rotating rings with a sustained aggregate data rate of 180 GB/s. Additionally, each chip has high-data-rate, low-latency memory and I/O interfaces.

The Cell Accelerator Board has 1 GiB of XDR DRAM with a peak data rate of 22.4 GB/s. In addition, the board has 4-GiB DDR2 DRAM attached to the companion chip. Both the XDR and DDR2 memory can be accessed via the Cell processor's direct memory access (DMA) engines, the companion chip's high- performance DMA engine, or an external PCI Express device or host.

This platform was used in the domain of digital cinema processing by the University of California at San-Diego [16, 97] and by the CineGrid [21] organization. In the last couple of years the interest in this platform is decreasing. IBM declared that it will no longer develop the 32 SPE variant of the processor, and it is increasingly harder to find new research projects using it. That is due to the complex programing model that it presents [76], and the very different scheduling mechanisms required for the SPEs to access external memory [13]. There are however, third party products that try to simplify it [82].

2.1.4. Storm-1

Storm-1 from SPI [111] combines data-parallel execution with a compiler-managed memory hierarchy with explicit data movement therefore exploiting data locality and parallelism to reduce global data rate needs [29, 30]. The execution model is similar to that of a traditional DSP with attention to performance-critical kernels. Main threads run on a 300 MHz RISC processor with kernel functions offloaded to the Data Parallel Unit (DPU). Kernels process and produce streams, which are arbitrary records of a finite size, such as a sequence of data blocks from a larger external array. This abstraction allows the compiler and hardware to efficiently manage data movement and kernel sequencing. The DPU executes one kernel at a time across up to 16 groups of five 32-bit ALUs running at 700MHz in a Very Large Instruction Word (VLIW) organization, processing the stream data in local memory. Kernels often form pipelines that share intermediate results, and overlapping stream loads and execution improves latency. It achieves 40 Mpix/s on motion estimation.

2.1.5. FPGA based processors

The invention of the FPGA is attributed to Ross Freeman (a co-founder of Xilinx Inc) in 1984 [127] (Patent No. 4,870,302). Today, Xilinx Inc. and Altera Corporation are the market leaders in fine-grained SRAM-based FPGA devices. A number of smaller companies focus on niche markets which include non-volatile FPGA memory (Lattice Semiconductors), flash-based FPGA memory (Actel), handheld applications (QuickLogic) and very fast, up to 1.5 GHz, FPGA devices (Achronix [3]).

FPGAs and ASSP (application-specific signal processors) have typically been used in dedicated hardware platforms, but with the emergence of new compilers the main CPU manufacturers have recognized the performance and power advantages of a hardware (FPGA, ASSP), software (CPU) approach. AMD pioneered this field with their Torrenza initiative [26] that allows other companies to develop and deploy application-specific co-processors to work alongside Opteron AMD processors in multi-socket systems. Intel followed shortly with their own standard: QuickAssist Accelerator Abstraction Layer (AAL) [79] that just like AMD's allows high speed communication via the front side bus between the main CPU (s) and the accelerator(s) [73].

2.1.6. FPOA

The Arrix FPOA (Field Programmable Object Array) [100, 124] from MathStar is a two dimensional array of 16 bit silicon objects, such as an Arithmetic Logic Unit, Multiply-Accumulator, Register File and Static Random Access Memory (SRAM). The objects and their interconnect are configurable and devices can contain up to 400 objects running at 1GHz. It also contains two external Synchronous Dynamic Random Access Memory (SDRAM) controllers with a peak data rate of 26 Gbit/s each.

2.1.7. Software on standard processors

The general purpose computing hardware most commonly used today is the mass produced PC. Its performance has been following Moore's law [83] and its predictable that it will continue doing so in the near future. This allows pure software solutions to increase their performance just by simply upgrading the PC platform that they run on.

Furthermore, pure software solutions have the advantage that there are many development tools available, many different programing languages and it is easy to find qualified engineers that can use them. In addition to that, software is easy to modify, adapt and extend to meet even fast changing project requirements.

A new development in recent years is the replacement of the single CPU in the mass produced PC with multi-cores and/or multi-processors [18]. This allows real parallel processing (8 and 16 core machines are now common) but the software needs to be re-written in order to use the available resources in an efficient way. For all Microsoft Windows operating systems the amount of supported physical processors is limited to two. However on newer Windows 7 systems each processor can contain up to 32 processor cores for 32-bit based systems and 256 processor cores for 64-bit based systems [2].

But for high resolution image processing applications its performance is still lacking when compared to other parallel oriented platforms. This is due to the sequential nature that software is run on those machines. Software companies are embracing the new possibilities and producing their software in such a way that it is scalable with the number of processors. It is

[2] http://www.microsoft.com/windows/windows-7/get/system-requirements.aspx

foreseeable that the performance gap between software running on multi-core processors and specialized parallel platforms will get smaller in the future.

2.2. FPGA programming methodologies

This section provides an overview of the FPGA programing methodologies available today.

Section 2.2.1 describes ways to program FPGAs using the C programing language and Section 2.2.2 using Matlab/Simulink. Both these methodologies are not flexible enough to develop the architectural features that we plan to achieve, and are only presented here for completeness sake.

We search a methodology that is flexible enough to implement the FlexWAFE architecture, namely:

- it must support hardware **modeling** and **verification** including the FlexWAFE software.
- it should semi-automate the **generation** and synthesis of the FlexWAFE architecture components.

These goals are achieved by a mix of scripts and VHDL code that will be presented in Section 2.2.3.

2.2.1. C as input

New developments in recent years have reduced the complexity of programming these systems by offering a high level programming environment that increases the programmer's productivity by increasing the level of abstraction at which they work. The Carte system [110] from SRC Computers [109] allows compilation of both standard C and FORTRAN code into an FPGA bitfile. It also allows debugging using the standard software debugging tools. Impulse C [92] is a similar product from *Impulse Accelerated Technologies* but it only accepts C as input language. There are an emerging number of vendors in this market [11], some of them, like Nallatech [84] provide not only a programming environment and development tools like DIMEtalk [85] but also hardware boards of which the DIME-II product line is an example [86].

The *FPGA High Performance Computing Alliance* promotes the use of Xilinx FPGAs in the field of HPC. The Trident [121] open source C compiler translates algorithmic high-level language code into hardware circuits.

AutoESL from Xilinx [14] is one of the newest of these products. It supports Xilinx architecture aware synthesis optimizations i.e. uses on-chip memories, DSP elements, MPMC and PLB IP components.

This methodology is good to develop a fixed algorithm, but it is not efficient at implementing the weakly-programmable features we seek.

2.2.2. Matlab/Simulink as input

Matlab and Simulink have established themselves as standard engineering tools for rapid prototyping on new algorithms and model based design. According to *The Mathworks*, the company that develops the software, Matlab is a text oriented software to perform mathematical calculations, analyzing and visualizing data, and writing new software programs. Simulink is a graphical oriented software for modeling and simulating complex and dynamic systems.

Both Matlab and Simulink were used during the preparation of this thesis to validate the results of the Very High Speed Integrated Circuit HDL (VHDL) simulations or the hardware results. A Matlab implementation of the Discrete Wavelet Transformation (DWT) based Noise Reduction (NR) algorithm was used first to develop the algorithm itself and later on to validate the FPGA results. The advantages of developing algorithms with Matlab is that it is easy to express even the most complex mathematical operations with it, and is easy to display the results especially in the field of image processing.

The downside of Matlab is its lack of speed. It interprets each single line of code each time it runs. There are ways of accelerating it by explicitly compiling the programs, but tests by [42] have shown that the results are an order of magnitude slower than hand coded C implementations. One other problem is that there is no automated way to generate code suitable for FPGAs.

Simulink on the other hand allows not only simulation on a standard PC but also automated generation of FPGA code. This is mostly possible due to the parallel and concurrent way that Simulink uses to represent systems. The automated translation tools do not need to transform sequential code (like in standard C or Matlab) into parallel code, the user already expressed the system behavior in a parallel way. There are several automated translation tools between Simulink models (.mdl files) and Hardware Description Language (HDL) (VHDL or Verilog files) for FPGA implementation:

- *Simulink HDL coder* from The Mathworks
- *Synplify DSP* from former Synplicity, now Synopsys
- *System Generator for DSP* from Xilinx
- *DSP builder* from Altera

All these tools provide a library of basic blocks that are mapped one-to-one into an HDL primitive like a register, or to some more complex FPGA primitive like an adder or an embedded multiplier block. FPGA vendors like Xilinx or Altera also provide parameterizable blocks that implement higher complexity operations like Fast Fourier Transformation (FFT) or viterbi decoders and that are optimized for their FPGA families. All these blocks are then to be instantiated and interconnected using Simulink's GUI.

During the course of this thesis, *System Generator for DSP* was used to implement a simplified version of the 1D DWT and its inverse. It operated as expected and it was possible to simulate it in pure software and latter run it in FPGA hardware. However, we noticed that the graphical nature of the Simulink modeling language made it somehow difficult to capture some of the proprieties of the hardware, namely variable bitwidth and some of the simple bit manipulation operations. Furthermore there was no way to physically constrain the location

of the generated code inside the FPGA (no floorplaning was possible). Again this methodology builds specialized hardware, and does not allow to implement weakly-programmable structures in an efficient way.

2.2.3. Hardware description languages

Hardware Description Languages (HDL) like VHDL [24] and Verilog were developed in the mid 80's, first as a means to document code requirements and intended functionality, and latter for simulation and for synthesis of logic circuits of FPGA or ASIC fabric.

One advantage of these languages is that they capture the full intent of the design, the developer controls every single detail of the implementation. When required it is possible to instantiate, parametrize and place FPGA primitive blocks allowing the designer to fully optimize the design for a particular FPGA family/technology. For instance it is possible to instantiate a block of RAM, parametrize its bitwidth, capacity, latency, amount of read and or write ports and define its placement inside the FPGA. Such an operation, however trivial as it may appear, is not possible with other languages like SystemC for example.

On the other hand it is possible to abstract the implementation details and concentrate on the functionality by using high level functionalities provided by the HDL. An example of those is state machine implementation. It's only necessary to describe the state transitions and the output equations, the optimal state encoding (Gray, Johnson, one-hot, binary, etc) is done by the synthesis tool.

It is however necessary to be an experienced designer in order to produce code that efficiently uses the underlying FPGA structures. The code can be simulated using software based simulators like Mentor Graphics Modelsim. Simulation can be run step by step and provides full visibility of all signals in the design, this simplifies debug. But it runs at only a fraction of the speed of the real FPGA and all of the FPGA I/Os need to be modeled in the simulation, there is no possibility to attach any I/O interfaces to the software simulation, this makes debugging of the entire system more difficult or even impossible. When running the code on the FPGA, full speed is achievable, I/O is available, but only a small subset of signals is visible, namely the ones available at the FPGA pins.

As seen above HDLs have advantages and disadvantages when compared to other programming techniques. We chose HDLs to describe our architecture because it allowed us to achieve a performance that we do not think is reachable with the other techniques. Some degree of flexibility is provided by the software controlling the weakly-programmable hardware structures.

2.3. Summary and Conclusion

All the technologies presented in this chapter map a single application to one hardware component (one-to-one mapping) or all applications to one hardware (n to one mapping).

The first category has been used extensively by our industrial partner [115], [119], [118] it delivers high-performance but is inflexible and too complex and time-consuming to develop, the second one is flexible at the expense of execution speed or power consumption.

The best solution would be to have multiple hardware components with just enough amount of flexibility such that, they could serve multiple applications but that each one would be specialized enough so that it could be resource and performance efficient. Application-Specific Instruction-set Processors (ASIP) [36] try to do just that, but they are nevertheless CPUs with hardware extensions and therefore typically as big, or even bigger than most general purpose CPUs. ASIPs need to have a CPU basis because the compilers need to be able to map any general purpose application into them in order to not restrict their application field. But that makes them big and in turn that makes it costly to build massive parallel systems with them. Their instruction-set is fixed and therefore it is not possible to customize it to adapt to customer specific needs. Besides that, high-data-rate optimized memory accesses are typically not efficient in these processors.

So we developed a set of optimized hardware components for one or more simple functions. They are basically ASIPs without the CPU part. We compose them in a stream oriented fashion and optimize the memory accesses and the data-flow manually using library functions. We tested this approach on some case studies that will be presented in chapter 6 on page 65. These show that our approach has advantages over the architectures and methodologies presented in this chapter.

3. FlexWAFE

This chapter describes the **Flex**ible **W**eakly-programmable **A**dvanced **F**ilm **E**ngine (FlexWAFE) in a bottom-up way, starting with the basic blocks and then explaining the hierarchy of blocks that control them. It starts by explaining the inter-block communication signaling in section 3.2 followed by a detailed explanation of the building blocks section 3.3 to section 3.6. An efficient multi-ported SDRAM memory controller (section 3.5) and a virtual multi-channel inter-FPGA communication infrastructure (section 3.6) developed by Dr. Sven Heithecker [49] are briefly described for completeness sake. The distributed control structures are presented in section 3.7. Finally section 3.8 summarizes and concludes this chapter.

3.1. FlexWAFE Reconfigurable Architecture

Hardware architectures are characterized among other metrics by their functionality, flexibility and inter-connectivity. In many systems the functional flexibility is achieved by an instruction set. General purpose processors have a complete instruction set, which guarantees high flexibility but comes at the expense of area and control complexity. Weakly-programmable architectures, on the other hand, offer a limited instruction set but can be realized using a simple interconnect and with small area foot print. The advantage is that the specialized data-paths are able to solve the computational critical parts in less cycles and with less area than is possible on highly programmable data-paths [87]. Furthermore they are advantageous in therms of interconnect, their simpler interconnect is faster and smaller [75]. Therefore, these architectures are especially suitable for image processing algorithms, which only require a small set of operations and whose interconnect can be easily expressed.

The weakly-programmable FlexWAFE architecture was designed for image stream processing, following a graph-based compositional design approach that will be detailed in Subsection 3.2. It consists of different types of components, which have been introduced in [37] and will be presented in detail in the following subsections. Essentially, in the FlexWAFE approach, weak-programmability is used to share the hardware for different functions, instead of complex FPGA dynamic (partial)-reconfiguration. The crucial advantage is a much faster context switch that takes a single cycle, with little overhead cost, and support for a very high area utilization.

In Figure 3.1, a generic example of the architecture is shown. One or more input streams enter the FPGA chip via the time division multiplexing (TDM) based stream demultiplexer (Subsection 3.6) in the left-side of the Figure, are processed by the Data Processing Units (DPU) (Subsection 3.3) along its data path, and finally leave the chip via the TDM based stream multiplexer in the right side of the Figure. The TDM based stream demultiplexer and multiplexer

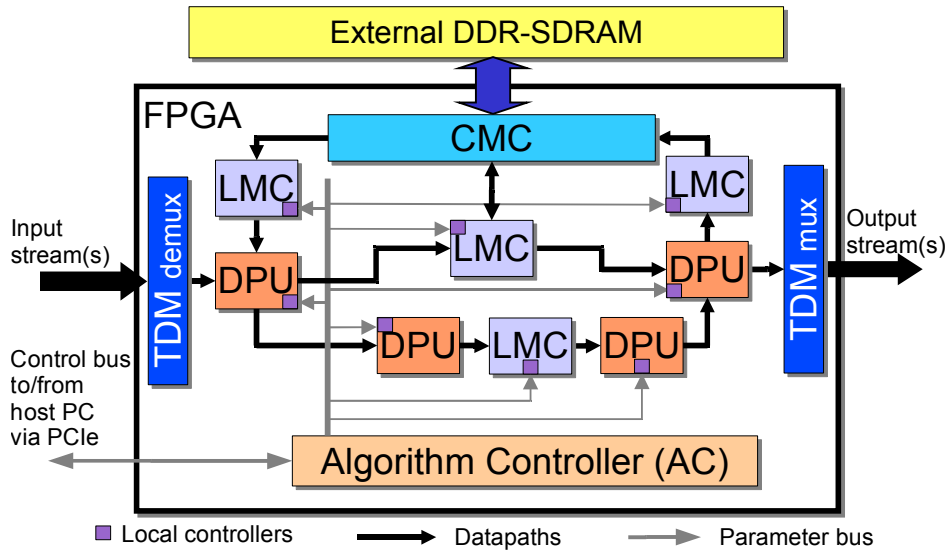


Figure 3.1.: Generic example of the FlexWAFE reconfigurable architecture

are optional, and are only used if more than one logic streams are used. A group of DPUs can be combined into a processing group (PG). A PG has the same interfaces as the DPUs but the formation of groups simplifies design reuse. The processing units are either connected directly to each other or are intercepted by Local Memory with Controllers (LMC) (Subsection 3.4) that provide memory services like reordering buffers, access at external SDRAM or scratch pad memories. Accesses to off-chip memory are realized via a custom memory controller (CMC in Subsection 3.5). The DPU and LMC modules are locally controlled by their respective Local Controllers (Subsection 3.7.1) that are weakly-programmable by an Algorithm Controller (AC) (Subsection 3.7.2), which sends run-time configurable instructions, thereby controlling the global algorithm sequence and synchronization. The algorithm controller itself is configured by a control bus (see Section 3.7.3).

All components are implemented as technology independent VHDL modules and are optimized for speed. Other optimization factors such as chip area or power consumption were not considered, as the targeted application domain of high-end image processing relies on large FPGAs, which are operated in standard workstations. The components can be parameterized in data word and address length, supported address, data functions, etc. via VHDL generics at design time. All VHDL modules were designed with portability in mind using techniques like inference instead of instantiation for specific FPGA parts like Xilinx block RAMs. This portability of the architecture was verified by porting to multiple hardware platforms as explained in section 5.4.

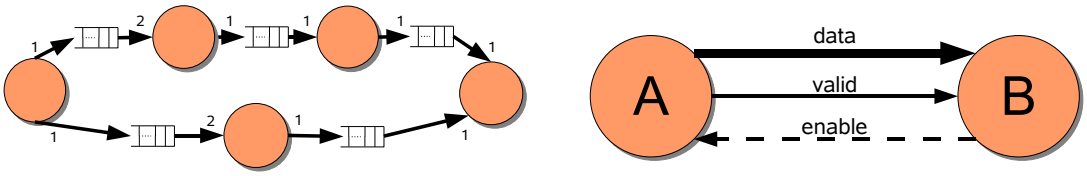
The described architectural approach is well suited for the processing of non data-dependent streams but can also be used for algorithms with data dependencies like the one shown in section 6.2.

3.2. Inter-block Signaling

The global data flow follows the synchronous data flow model (SDF, [69, 74]): the system is modeled as a directed graph, where the nodes represent processing elements and the arcs represent communication channels for data tokens. In the general SDF model, arcs are communication channels with infinite buffer length. In the FlexWAFE implementation, the length must be adapted to the application's real needs and is user configurable. Node execution is enabled if enough tokens are available at their input arcs and uses a technique called back pressure to avoid buffer overflow. FlexWAFE supports colored tokens by attaching control information together with the token data.

There is nothing really new about our implementation of SDF but for the sake on completeness we will describe our implementation in the following paragraphs. The knowledgeable reader might want to skip directly to section 3.3.

This model was implemented via the use of a *request-acknowledge* protocol between the processing elements. Only three signals are used for each graph arc, which simplifies design and routing while still allowing to express the dependencies of the graph. The signals can be



(a) Example of a standard SDF graph with infinite buffers in the arcs

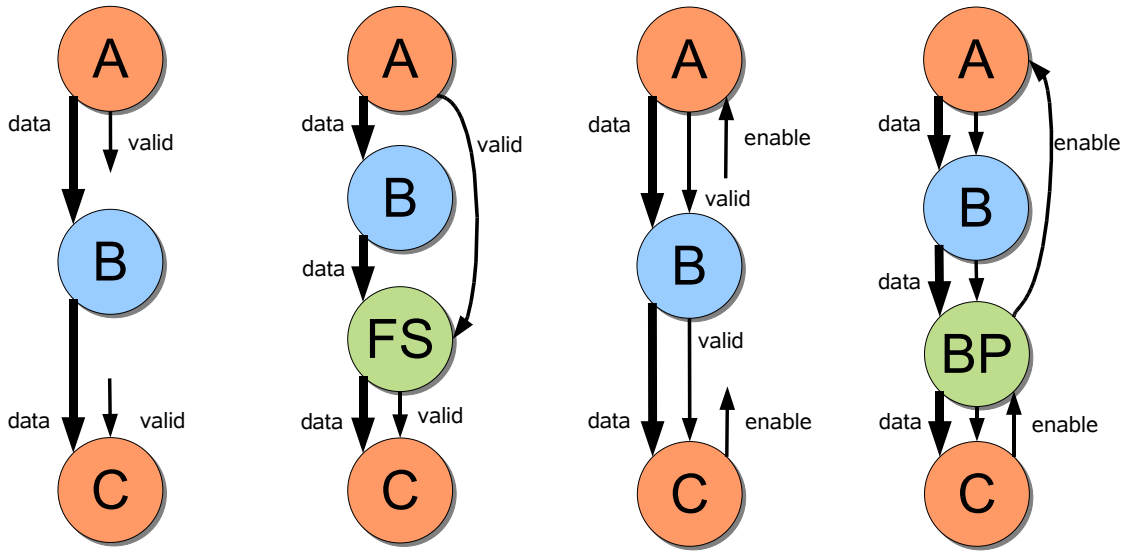
(b) FlexWAFE handshake implementation of a graph arch between two graph nodes

Figure 3.2.: SDF graphs

seen in Figure 3.2b and are: *data*, the data token to be processed; *valid*, the validity of the data token and *enable*, to signify that the receiving block is ready to receive a new token. *valid* and *data* have the same direction as the arc, *enable* has the opposite direction. The meaning of these signals is a bit different from the traditional req-ack signals, therefore we decided to use valid-enable to avoid confusion.

Available data at processing element inputs is signaled by *valid* signals; data is processed only if all inputs are valid. With this technique, a blocking-read behavior is implemented. Consuming of tokens is indicated by the *enable* signal, which in turn lets the preceding processing element produce a new data token, thereby realizing blocking write transactions. This *back pressuring* of the *enable* signal to the preceding blocks eliminates the need for sufficient buffering at every arc and allows smooth and reliable starting and stopping of each node. Therefore, each arc has a buffer size of zero tokens.

When designing a new system, there might be a need use existing IP blocks that do not have *valid* and *enable* signals (blocking-read/blocking-write behavior). The FlexWAFE library addresses this issue by providing two blocks that add this functionality. This is shown in Figure 3.3a: block B shall be integrated with block A and C from the FlexWAFE library, has a processing latency of n clock cycles, and does not provide a *valid* control output.



(a) (b) (c) (d)

Figure 3.3.: (a) Block B does not provide a *valid* signal; (b) Block FS allows blocking-read operation of block C; (c) Block B does not provide a *enable* signal and; (d) Block BP allows blocking-write operation of block A

The library provides a *forwardsync* block (block FS in Figure 3.3b) that produces the missing *valid* control output, based on the n constant latency parameter of block B. It delays the *valid* input n cycles to produce the *valid* output. If the latency is not constant and known beforehand, the user must write his own *forwardsync* block.

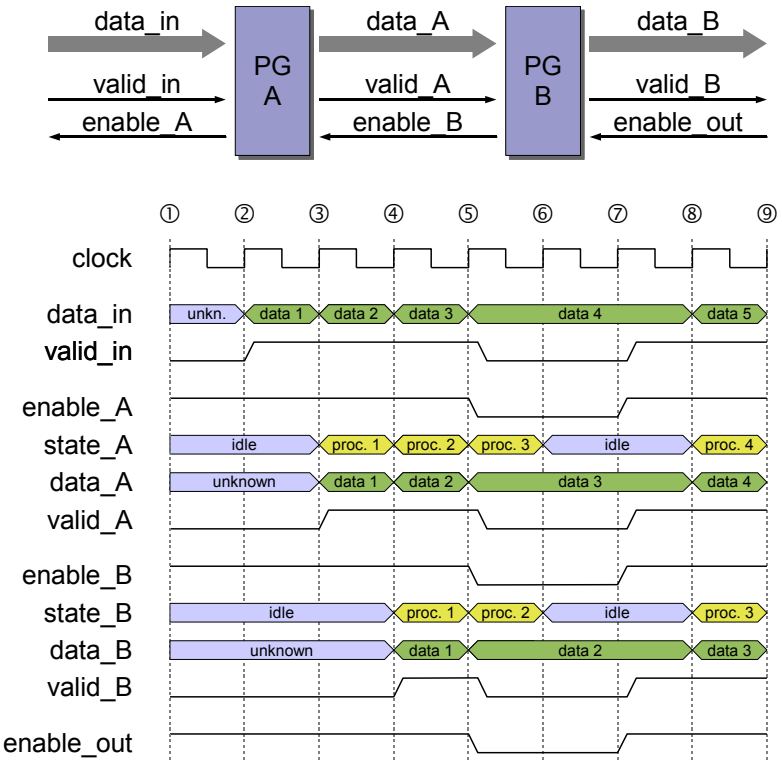
Block B in Figure 3.3c is an example of a block that does not provide the *enable* signal. To integrate it a *backpressure* block can be used, which again is configured with a parameter n that is the maximum latency of block B. This extra block is a write-through optimized first-in-first-out (FIFO) of depth n and is depicted as block BP in Figure 3.3d. As soon as block C disables its input, BP captures the data continuously sent by block B and disables block A, thereby stopping the preceding processing chain.

The *forwardsync* and the *backpressure* blocks can transform an existing non-blocking blockset into a blocking-read/blocking-write system by adding simple, configurable, non-intrusive blocks that have been optimized for area and speed although written in technology independent VHDL.

Figure 3.4 on the facing page shows an example with a chain of two DPUs or PGs. Both these units will be explained in detail in section 3.3.

3.2.1. Data Types

Data is transported from PG to PG by bit-parallel transmission of data words which come in a variety of formats.



1. No data available at input, both PGs are idle; all valid signals deasserted, all enable signals asserted.
2. Valid data 1 at `data_in`, `valid_in` signal asserted.
3. PG A registers data 1, starts processing and outputs processed data 1 n `data_A` (processing output latency not shown). PG A also asserts `valid_A` to signal to PG B the availability of data. Meanwhile, a new data word 2 appears in `data_in`, therefore `valid_A` is kept asserted.
4. PG B processes data 1, PG A processes data 2, new data word 3 available at input. All valid-signals remain asserted.
5. PG B processes data 2, PG A processes data 3, new data word 4 available at input. However, the output is not capable of processing more words and deasserts `enable_out`. PG B immediately deasserts `valid_B` and `enable_B`; `enable_B` is forwarded to PG A which deasserts `valid_A` and `enable_A`. Deasserting `enable_A` stops the input from producing more data.
6. The complete system is halted.
7. The output asserts `enable_out`, which leads to an assertion of all remaining valid- and enable-signals.
8. The system continues processing (PG B processes data 3, PG A processes data 4, new data 5 available at input).

Figure 3.4.: Simple datapath communication example using two processing groups

For example, take the DWT filter stages of the noise reduction application presented in the introduction. Since they operate using integers, to maintain accuracy, the bit resolution needs to be increased at each filter stage by 4 to 8 bits. So when starting with 10 bit words[1], a 3 level DWT will result in output word sizes of 14 to 30 bits (see Figure 6.13 on page 86). Another example is the Motion Estimation (ME) part of this application. It's input pixel stream format has a word size of 32 bits (one pixel, RGB 10 bit per color channel, 2 bit for frame and line synchronization), while the generated Motion Vectors (MV) have a word size of 8 bits (2×4 bits integer, vertical and horizontal motion distance -8 ...+7).

However, following the SDF paradigm, the kind of token is defined at design time and remains constant[2].

For the implementation of the communication channels only the *physical format* (bit width) of the tokens is important, not the logical format (the meaning of each bit inside the token). It has, of course, to be ensured that only compatible PGs are connected.

3.3. DPU- Data Processing Unit

The functionality of an image processing application is implemented by partitioning the application into data processing units. These units (graph nodes in the previous subsection) execute the computational tasks and are equipped with multiple input and output ports (graph arcs in the previous subsection), which allow the realization of blocks with generic complexity. Furthermore, DPUs typically possess local cache-like memories to accelerate their operations, thereby increasing the overall system's performance.

A typical example for a DPU is a Finite Impulse Response (FIR) filter depicted in Figure 3.5, which is a common element of image processing algorithms. The filter coefficients are configurable by the Local Controller (see Section 3.7.1) and can be exchanged during runtime, which allows the implementation of adaptive filters. Such filters are useful when wildly divergent images (e.g. scene changes between day and night) require different filter coefficients.

Some DPUs include information for relative placement on the FPGA. This allows the construction of computing grids of simple processing elements without floorplanning each DPU separately. The following list shows some DPU examples with varying complexity that have been developed for the FlexWAFE library:

- adder - adds two streams of signed or unsigned integers and produces a result stream
- RGB-> YCbCr - converts a stream of Red, Green, and Blue pixel color information into the YCbCr color space according to SMPTE (Society of Motion Picture and Television Engineers) 274 and 296 standards.
- soft shrinkage - input values smaller than a specified threshold get converted to zero, and bigger ones get the threshold value subtracted from them

[1] one pixel, 10 bit single color channel

[2] In opposition to changing for example the resolution during run-time, however this is not covered in this thesis. The format specifications can be seen as maximum values.

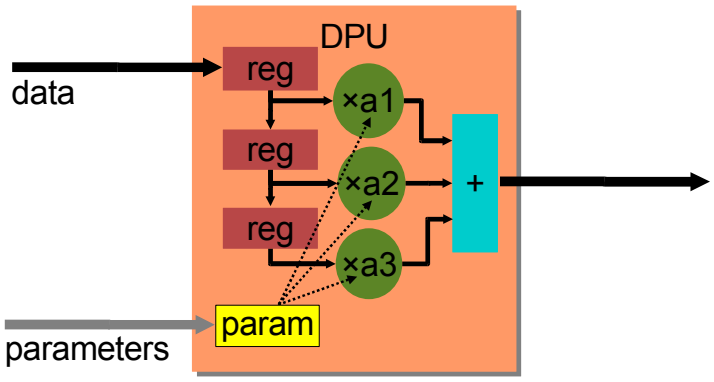


Figure 3.5.: A FIR filter DPU

- discrete wavelet transform - one dimensional and two dimensional direct and inverse transformation into wavelet space
- histogram - one dimensional and two dimensional discrete bin based histogram with configurable image tile size and bin size
- motion estimation - two dimensional full pixel matching based motion estimation

As the DPUs highly depend on the underlying algorithms, the implementation of new DPUs will be required when realizing new algorithms for the FlexWAFE library.

3.3.1. Processing Groups

Some applications can use a large number of similar DPUs cascaded with each other. To ease the design, and minimize routing congestion, these DPUs should be kept simple. For simplification, they do not use the back-pressuring signaling scheme described in 3.2 in the form of the enable signal. Instead, they are grouped together and a BP block is placed at their output, providing the necessary back-pressuring functionality in a centralized way, instead of distributed small buffers inside each DPU. This group is then called a Processing Group (PG) and has the same valid-enable interface as a standard DPU.

3.3.2. SIMD like processing

Vector and multidimensional signal processing require that the same operation should be performed on multiple data tokens. On general purpose processing platforms this is know as Single Instruction Multiple Data (SIMD) and some processors are optimized for this kind of operation. Such an approach reuses the synchronization and control logic to control multiple data processing paths. In the FlexWAFE architecture, a similar approach was taken for some of the DPUs. They were designed to split the input tokens into sub-tokens, perform the same operation on all of them and reassemble an output token from the results. This approach comes at the expense of a few extra code lines per DPU but can save a lot of logic because it explicitly shares the control logic. The synthesis tools sometimes do not optimize away similar control

paths and this methodology warranties that no duplicated logic occurs. This was one of the optimizations used on the lifting DWT implementation and it partially contributed to the 75% area reduction when compared with the FIR DWT implementation (section 6.4 on page 97).

3.4. LMC- Local Memory with Controller

As described in section 3.2, the communication channels of FlexWAFE have a buffer size of zero tokens. For buffering, explicit buffer nodes are introduced in the graph, which can have extra functionality, like synchronizing multiple streams, reordering streams, delaying streams, etc. These extra functionalities will be the topic of this subsection. The reason for using these blocks is design reuse. They can be integrated at any point of the signal processing data-path and typically have weakly-programmable capabilities that give them flexibility to change their function at run-time. Like the DPUs, they are fast, since they are tightly coupled with the distributed memory that populates any modern FPGA. This memory is accessed with programmable access patterns generated by weakly-programmable address generators.

There are several LMC types. The simplest, *lmc_fifo*, is a FIFO whose length and datawidth are configurable at synthesis time.

The next subsections will present the building blocks of the LMCs, starting by the simplest and ending with the most complex. Special detail is given for the address generators as those are play a key role in the LMCs flexibility.

3.4.1. Asynchronous FIFOs

Asynchronous FIFOs are an important basic building block, we took a look at the state of the art implementation [28], and decided to improve it. We replaced the binary counters with direct gray code counters, this improved the max. operating frequency by aprox. 10% when compared to [28]. We also added auxiliary state counters to simplify applications where data bursting and/or de-bursting is required. When very deep FIFOs are required, the FPGA on-chip memory amount is not enough. For those cases we developed two LMCs with access to external SDRAM that will be presented on Section 3.4.6. Appendix C on page 113 gives more details about the FlexWAFE implementation of asynchronous FIFOs.

3.4.2. Address Stepper

The function of this sequencer is rather simple and its structure is shown in Figure 3.6. This component is basically a flexible, loadable counter based on [47]. It has a *start* input to set the counter's starting point, loadable by the *nload* signal; the value that is added in each counter step is controlled by the *delta* input and the counter limit is set by the *end* input.

Because this component is meant to be small and fast, there is no overflow protection mechanism. It is up to the user to react promptly to the `done` signal and reconfigure a new sequence at the inputs of this component. Since all inputs are implemented as two's complement signed

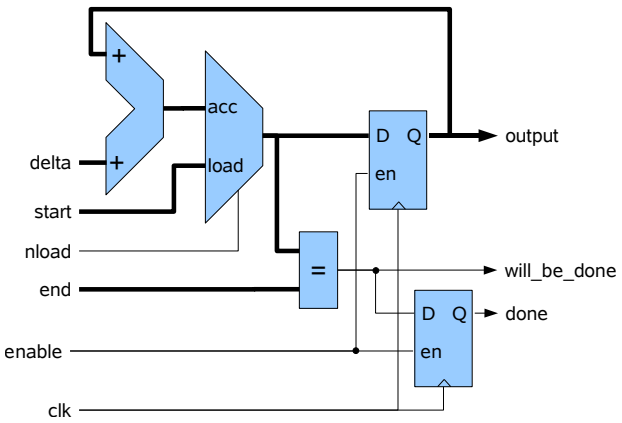


Figure 3.6.: Detailed functional stepper diagram

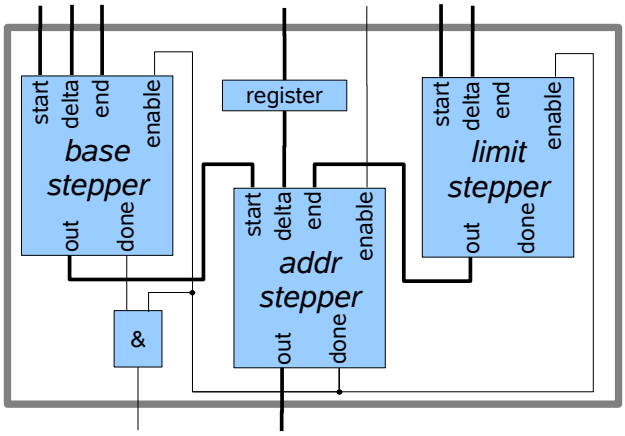
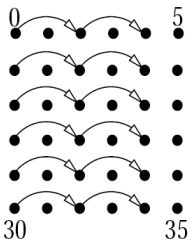


Figure 3.7.: Cascaded Stepper functional diagram

values, positive and negative numbers can be used. It contains two architectures: one generic register transfer level (RTL) implementation that can be used in every FPGA family, and one Xilinx Virtex II Pro optimized architecture that uses less resources and is faster because it combines the functionality of a full-bit adder and a multiplexer in a single Xilinx Virtex II Pro slice. To achieve this, the *load* signal needs to be active low and therefore it is named *nload*.

3.4.3. Cascaded Stepper

The cascaded stepper consists of three connected steppers, a register to buffer one incoming parameter and some control logic as shown in Figure 3.7. The Cascaded Steppers are based on those described in [47, 93] and generate address sequences using only six parameters. This set of parameters, although small, provides a large number of patterns, allowing the LMC to rotate, flip, decimate, extract a ROI (region-of-interest), scan in different zig-zag patterns or implement a combination of any of these operations.



parameter	value
base_start	0
base_delta	6
base_end	30
addr_delta	2
limit_start	4
limit_delta	6

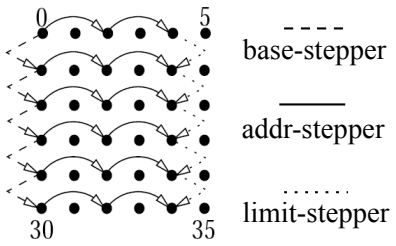


Figure 3.8.: Simple address pattern, horizontal decimation by two, on a 6x6 pixel picture (left); the parameters that generate it (center) and its generation via the 3 steppers of the Cascaded Stepper module (right)

The two outer steppers, named base- and limit-stepper, generate the parameters for the addr-stepper. The example pattern shown in Figure 3.8 will be used to explain the function of this structure. The desired address pattern is a simple walk-through of a 6x6 picture, where every even value is read (0, 2, 4, 6, ... 34). The base-stepper calculates the address of the first pixel of every line, whereas the limit-stepper generates the address of the last active pixel in each line. Those results are passed on to the addr-stepper, which generates the final output. For the pattern in Figure 3.8, one set of six parameters is needed. This set includes one complete set of 3 stepper parameters for the base-stepper. The limit-stepper requires one parameter less, because it runs in sync with the base-stepper, so no end parameter is required. The parameter set is completed by an addr_delta parameter for the addr-stepper. This stepper does not need any other parameters, since these are provided by the two outer steppers. This leads to the parameters in Figure 3.8 (center).

3.4.4. LMC reorder streams

To reorder a stream, an LMC address pattern transformer (*lmc_apt*) consists of one local memory and two Cascaded Steppers (one for the ingress, one for the egress stream address), each coupled with a counter that controls the amount of times the address pattern repeats itself. The local dual-ported memory is divided into m regions ($m = 2^k, k \in \mathbb{N} \wedge k \geq 1$), which allows a parallel operation of both address steppers in different regions. Each Cascaded Stepper accesses these regions sequentially and is guarded from the other stepper by hardware mutexes to exclude memory hazards. Basically, the egress can only access memory region n ($0 \leq n \leq m$) after the ingress finishes writing its pattern to it, and vice-verse; this forms a memory based circular buffer (see Appendix B on page 111). The memory's depth, width and number of regions are configurable by generics, and the memory consistency is kept automatically, making this block very flexible and easy to program. An example of an application of such an LMC is creating a block-based zig-zag stream: the input stream is stored sequentially block by block in the dual-ported local RAM and is read out using three patterns, whose parameters can be seen in Figure 3.9 (right). Ingress and egress sequential pictures are shown together with the calculations required for a generic square block size.

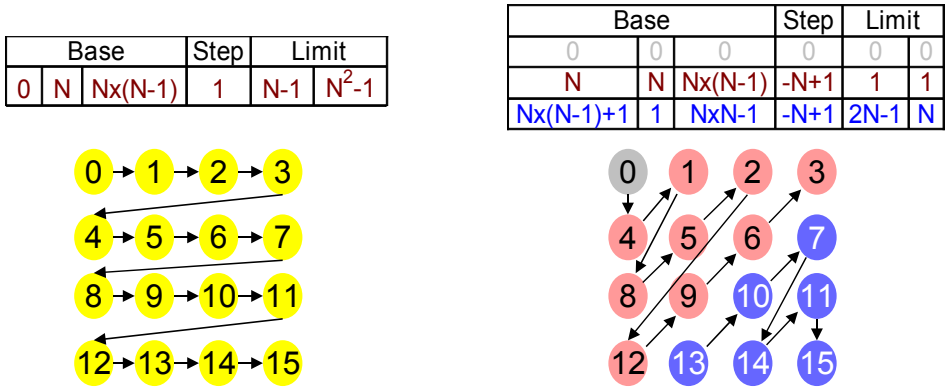


Figure 3.9.: Zig-zag address sequence example for image blocksize of 4. Cascaded Stepper parameters for ingress AG (left) and egress AG (right)

Transformation	Base			Step	Limit	
0° rotation	0	N	N×(N-1)	1	N-1	N^2-1
90° rotation	N-1	-1	0	N	N^2-1	-1
180° rotation	N^2-1	-N	N-1	-1	N×(N-1)	-N
270° rotation	N×(N-1)	1	N^2-1	-N	0	1
transposition	0	1	N-1	N	N×(N-1)	1
90° rotation + transposition	N-1	N	N^2-1	-1	0	N

Table 3.1.: Examples of access pattern transformations using the lmc_apt block

An other example of a reorder application is a 90 degrees rotation of an image block of n by n pixels. This operation is required in the motion estimation algorithm described in Section 6.2.1 for example.

First a block of image pixels is presented at the input of this block in a image line by image line manner. Such an image stream can be the output of a DPU or obtained by using an *LMC with external memory* presented in the section below. That stream is stored pixel by pixel on consecutive addresses of one of the m regions explained above. To do so the pattern presented in the left side of Figure 3.9 is used on the ingress address generator. The egress address generator is then programmed with one of the parameter sets presented in Table 3.1. As seen in the last table line, transformations can be combined to produce the desired result. The examples presented are for square image blocks, but rectangular image blocks are also possible.

3.4.5. LMC with external memory

Some image processing algorithms require more memory than the one available inside the FPGA, so external memory must be used. For that purpose, another set of LMCs was created. They consist of a Cascaded Stepper to generate addresses for the external memory and a local FIFO. Accessing external SDRAM memory is burst oriented (n consecutive data tokens)

which require only the data burst start address. The address of the following $n-1$ data tokens is automatically calculated by the SDRAM itself. This saves data-rate from FPGA to SDRAM because only the first of the n addresses is transmitted. In our system $n = 16$, so the address generators step forward in address steps of $n = 16$ as described in [125].

One LMC is used to write to external memory (LMC_S2C), another is used to read from it (LMC_C2S). This way input pattern sequences are independent of output pattern sequences and can be (re-)programmed on the fly to meet application demands.

3.4.6. Large FIFO based on external SDRAM memory

For applications where large FIFOs are required, an LMC exists that uses an external memory address region as storage space. It needs two address generators to access it (one for the read address sequence, one for the write address sequence), two small local FIFOs (again one for the incoming, one for the outgoing data) and some full-empty control logic. The full-empty logic takes into account SDRAM issued requests that have not yet been completed and the state of both local FIFOs.

3.4.7. Other LMCs

Some other simpler LMCs where built. The *lmc_sync_join*, is capable of synchronizing n multiple streams, each having its own datawidth. It stores incoming streams in independent token buffers. As soon as all buffers have at least one token, a combined token is written to the output. For this purpose it has n independent write pointers and one common read pointer.

By combining a *large FIFO based on external SDRAM memory* or an *asynchronous FIFO* with a *lmc_sync_join* a *lmc_line_delay* was built. Delaying one or more image lines is a common operation in image processing algorithms that requires storing the content of one or more image lines and output that content exactly one or more lines later synchronously to another pixel stream. When this other pixel stream produces new pixels, the line delay must also produce outputs; when the other pixel stream stalls, the line delay must also stall.

Both these LMCs have been implemented as non weakly-programmable because their parameters cannot change at run-time.

3.5. Custom DDR-SDRAM Memory Controller

As previously stated, the FlexWAFE architecture utilizes external DDR-SDRAM memory to store image data. Memory is clocked at 125 MHz (in the FlexFilm board implementation), allowing peak performance of 8 GBit per second. To achieve average rates close to this theoretical maximum, an access optimizing memory controller (CMC) depicted in Figure 3.10 was developed by a colleague and is thoroughly described in Chapter 5 of his PhD. thesis [49], [53] and [50]. A brief overview of this controller is provided in this section for completeness.

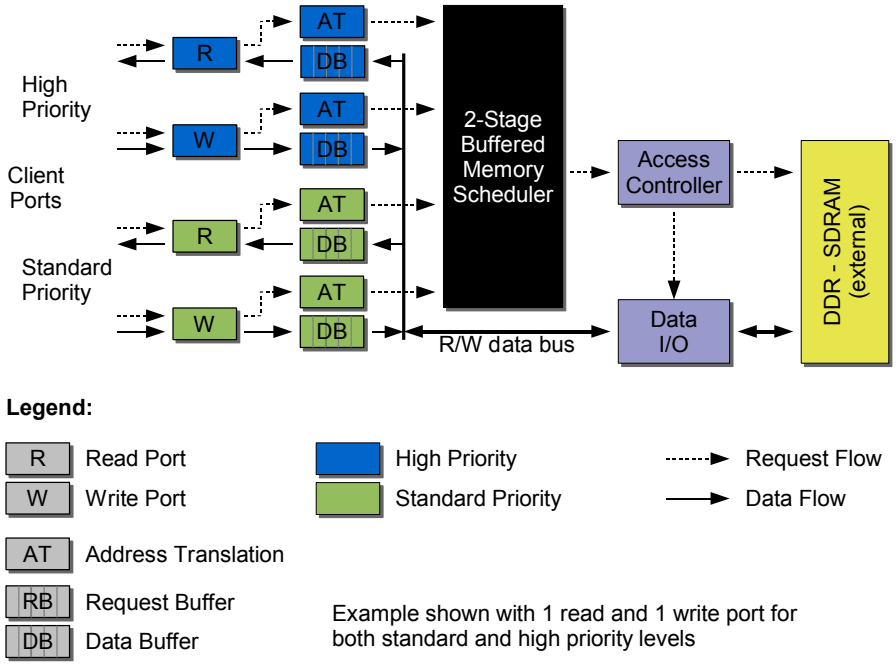


Figure 3.10.: SDRAM controller architecture

For real-time digital film processing, the FlexWAFE architecture requires data rates that cannot be delivered by simple memory controller designs offering first-come first-served access. To increase memory throughput utilization, memory access optimizations must be performed to reorder memory requests. This allows the memory controller to take advantage of the structure of SDRAM memory, which has a buffered parallel architecture that is organized into independent memory banks. The core of the controller, the two-stage buffered memory access scheduler, provides port based QoS queuing, performs access optimizations and issues requests to SDRAM. The average and worst case memory access latency depends on the used SDRAM characteristics, the number of ports and their priority.

The CMC remains flexible in terms of configurability, and has been designed to be compatible with other high-performance reconfigurable FPGA and ASIC platforms. Only a selected group of the core SDRAM controller components have been described here.

3.6. Inter-chip and Inter-board Communication

This task was done by a colleague and is thoroughly detailed in Chapters 3 and 4 of [49]. A simplified overview is given in this section.

Because the architecture is designed for stream processing, a reliable transport for multiple image streams between the FPGAs needed to be designed. Latencies should be kept at a minimum, since large latencies require large buffers, which have to be implemented inside the FlexWAFE FPGAs. Since the streams (currently) are periodic and their data rate is known at

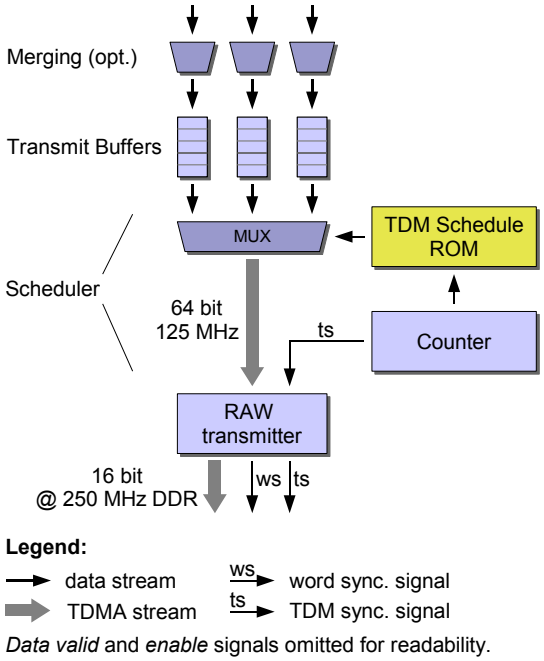


Figure 3.11.: Chip-2-Chip transmitter

design time, TDM[3] (time division multiplex) scheduling is a suitable solution. TDM means that each stream is granted access to the communication channel in fixed slots at fixed intervals. The communication channel works at a "packet size" of 64 bit. Figure 3.11 shows the communication transmit scheduler block diagram. The incoming data streams, which may differ in clock rate and word size, are first merged and zero-padded to 64-bit *raw* words and then stored in the transmit FIFOs. Each clock cycle, the scheduler selects one raw word from one FIFO and forwards it to the raw transmitter. On the physical layer, 16 parallel low voltage differential signaling (LVDS) connections are driven at quad data rate (QDR) (quad data rate, 4x125 MHz). For word synchronization and back pressuring, sideband control signals are used.

3.7. Control and Programmability

As seen in the previous subsections, the FlexWAFE architecture consists of stream oriented datapaths where some of the blocks have parameter inputs that can be changed at run-time. To achieve flexibility those parameters need to be programmable, preferably in a way that allows the parameters to change immediately once a certain condition is triggered. To achieve this, a global control architecture, coupled with loosely coupled local controllers was developed.

This hierarchical control architecture is shown in Figure 3.12 and briefly consists of:

- Host PC: Using a (optional) GUI the programs that the Algorithm Controllers (AC) (Section 3.7.2) will run are selected and later transferred to their respective ACs via the

[3] also referred as time division multiple access (TDMA)

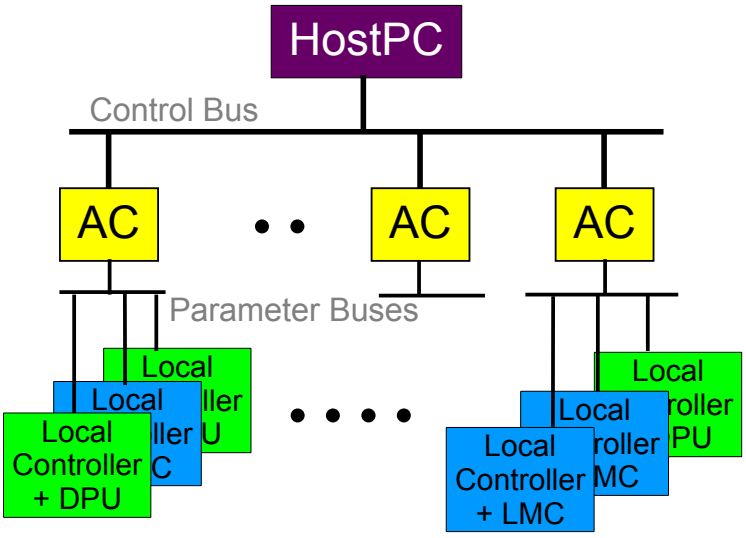


Figure 3.12.: FlexWAFE distributed control hierarchy structures

control bus (Section 3.7.3). This happens only once during initialization. Simplified host PC pseudo code:

```
foreach AC do
  // simple memory mapped transfer of a single contiguous data block
  program it via control bus
end
reset all ACs
```

- AC: Controls a complex datapath consisting of multiple LMCs and DPUs, each with it's own Local Controller (LC) (Section 3.7.1), that implement a function or algorithm (hence the name). The transmission of instructions that the AC sends to the LCs via the parameter bus is a *shadow operation* that happens in parallel with execution of other instructions (pipelining at block level). The AC synchronizes multiple LCs using barriers. Simplified AC pseudo code:

```
if (instruction == send)
    // simple memory mapped transfer of a single data token
    send data to LC via parameter bus
if (instruction == wait)
    barrier wait for one or more LC's done signals
if (instruction == goto)
    jump to goto address label
execute next instruction
```

- LC: Controls a single DPU or LMC. It parametrizes them and instructs them what to do and when to do it. It contains the shadow memories that allow the asynchronous transmission and parallel processing of parameters mentioned above and is tightly couped with the block it controls using handshake synchronization. The functionality of this block is described in Figure 3.16.

Weak programmability is achieved by a small controller, local to each block, running its program from a local memory. Since each block can be equipped with its own Local Controller/memory pair, the overall system's control is truly distributed and parallel. The program sequences run in these Local Controllers are written from Algorithm Controller (AC)s that are themselves small controller units. There can be more than one of these per FPGA and their programs are in turn written by the host computer. Control is therefore hierarchical, as shown in the example in Figure 3.12, Figure 3.13 and Figure 3.14. Essentially, in the FlexWAFE approach, weak-programmability is used to share the hardware for different functions, instead of complex FPGA dynamic (partial)-reconfiguration. The crucial advantage is a much faster context switch that takes a single cycle, with little overhead cost (two slices per stored bit), and support for a very high area utilization. On the other hand, flexibility is limited to the predefined functions. This loss in flexibility, however, can be compensated by a suitable collection of functions that will be presented later. Each of these controllers and the buses that interconnect them are explained in the following paragraphs.

3.7.1. Local Controller

A Local Controller (LC) is a tiny parameter/instruction sequencer tightly coupled with a local memory. It has only two instructions: *load* and *reset*. A *load* instruction outputs a set of parameters whose values are part of the instruction itself to an attached DPU or LMC. These multiple parameters are outputted simultaneously in a parallel fashion and can be seen as a single VLIW[4] word that controls one or more DPUs and/or LMCs. Afterwords, it waits for a *done* signal from the block it controls and steps to the next instruction. If that instruction is a *load*, the process repeats itself; if it is a *reset*, then it steps back to the very first instruction in memory as seen in Figure 3.16.

All parameters stored in the VLIW words are read from local memory in parallel (clock synchronously in a single cycle), but are written to local memory independently from each other via a parameter bus depicted in Figure 3.15. This allows the Local Controller to output one VLIW while other(s) are being written. This is necessary, for example, for the zig-zag pattern shown in Figure 3.9, where it avoids address generation interruptions by providing the next parameter set instantly. The AC programs all the three necessary parameter sets once, and the local controller runs them autonomously in loop, without a single idle clock cycle.

The number of parameters/instructions in the VLIW words and the width of each subword is configurable via VHDL generics. The FlexWAFE framework automatically derives the values of these generics from a Extensible Markup Language (XML) program written by the user. This program will be explained in detail in section 5.2. On Xilinx devices, the local memory is translated to look-up-table (LUT) based dual-ported distributed memory of depth 16 that exists in every slice and will therefore be placed together with the datapath unit that it controls. It is

[4]The VLIW term is used here in the Multiple Instruction Multiple Data (MIMD) sense. The instruction execution latency, is however, not known at compile time, and the instructions are not statically scheduled by the compiler like in a VLIW processor. Here the schedule is fully dynamic: an external *done* signal marks the completion of one instruction; after it, the next instruction starts execution.

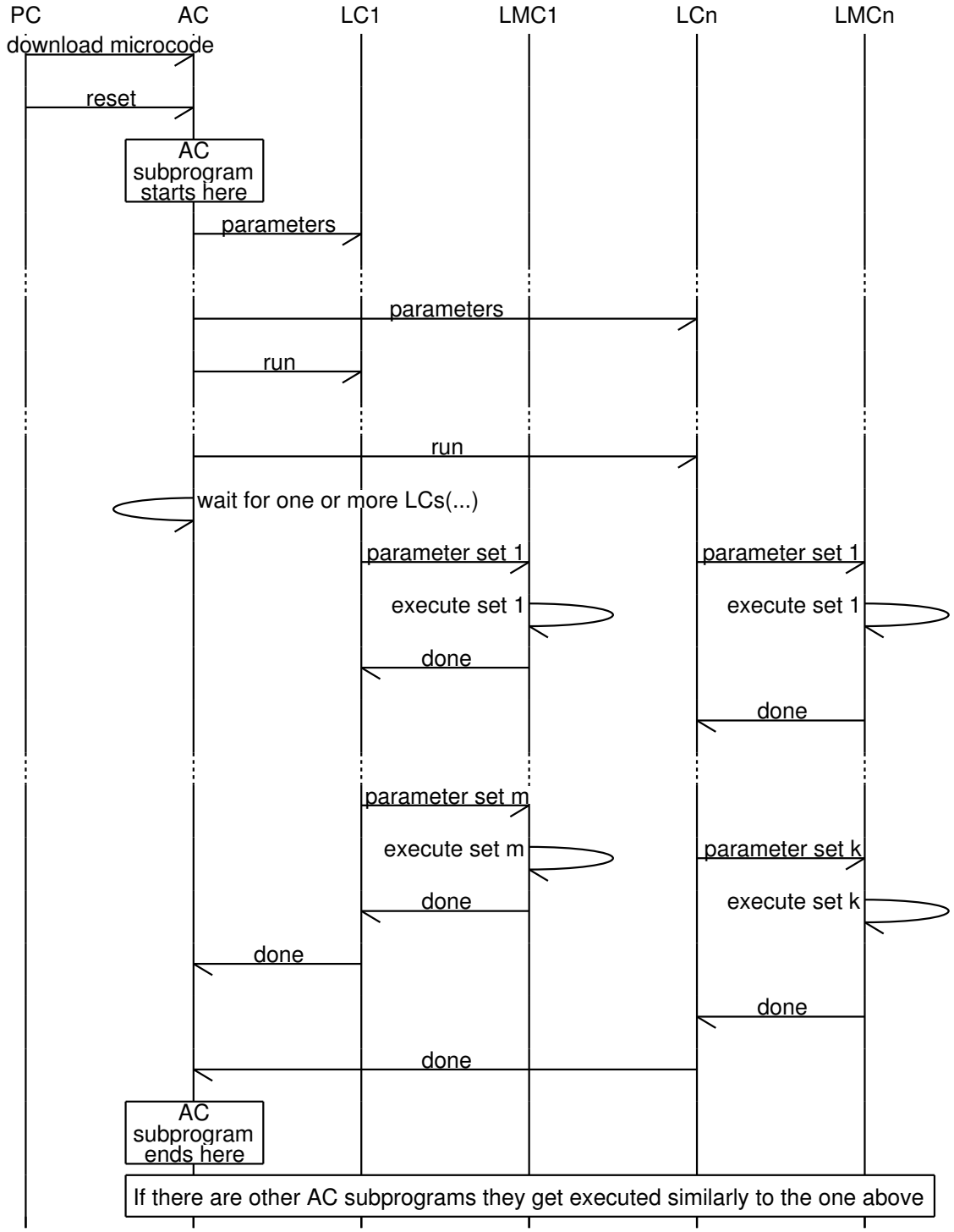


Figure 3.13.: FlexWAFE control hierarchy message sequence chart example

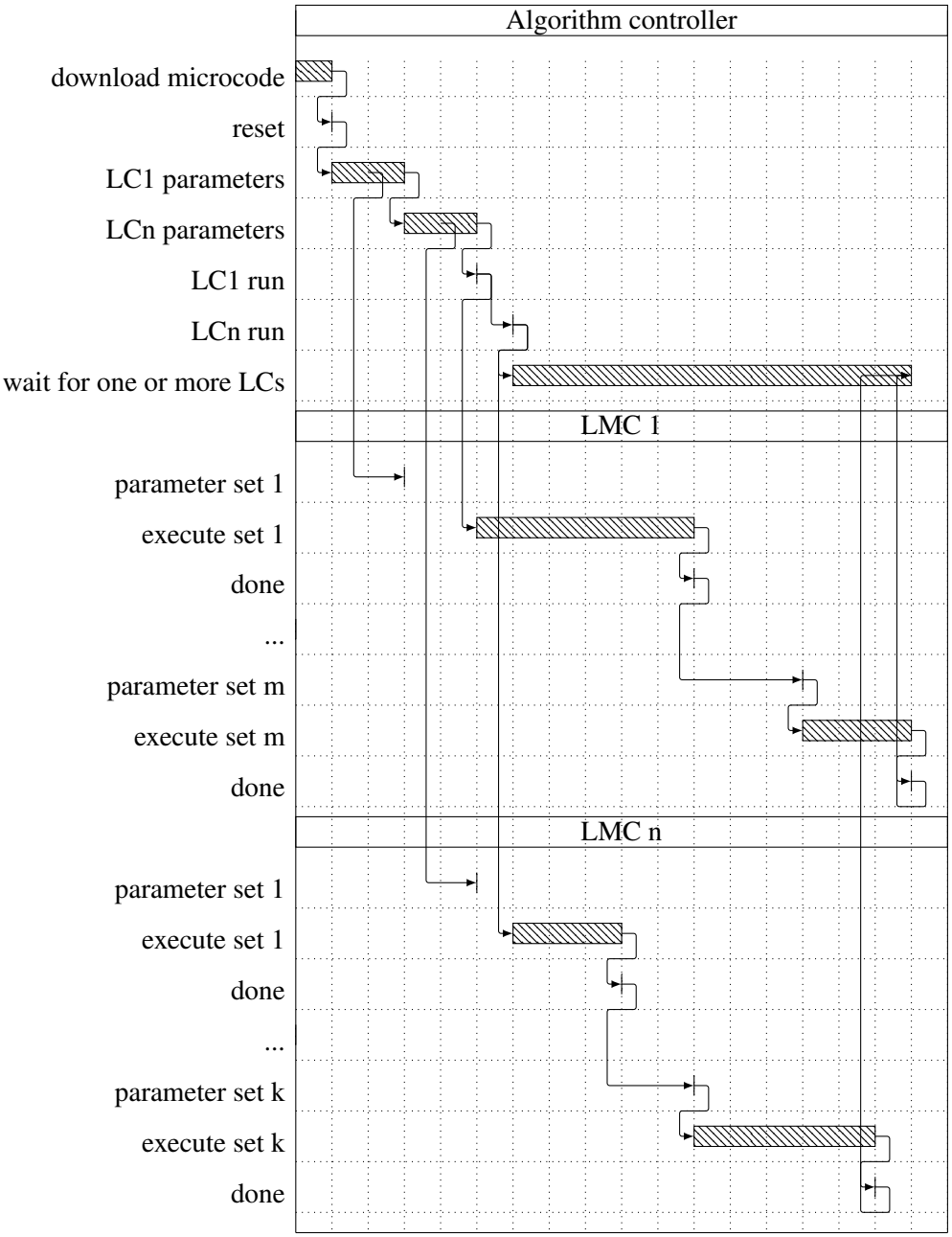


Figure 3.14.: Gantt chart showing the parallel execution of n LCs with their respective LMCs for the example presented in Figure 3.13

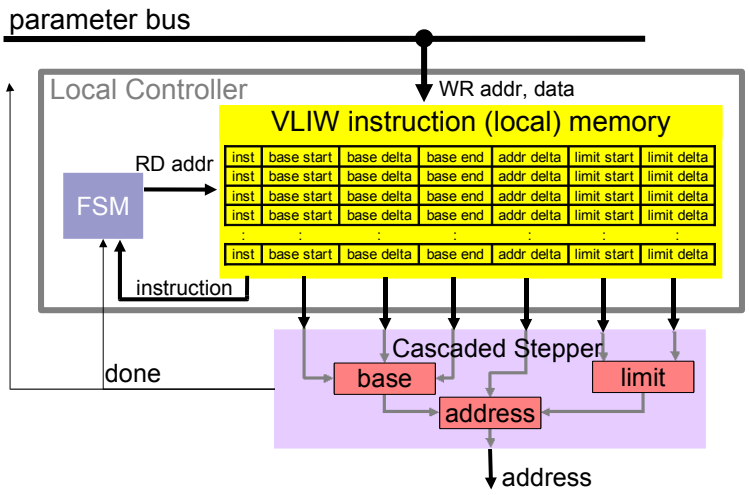


Figure 3.15.: Example of a Local Controller, attached to a Cascaded Stepper

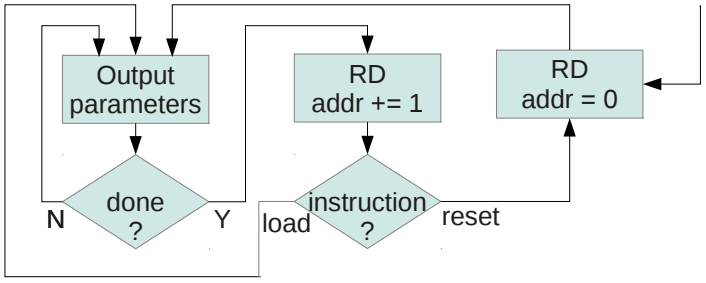


Figure 3.16.: Local controller state machine, *done* is an input and *instruction* is a flag in local memory

able to feed the datapath with one new set of parameters per clock cycle. Figure 3.15 shows an example of tight integration of a Local Controller and a *Cascaded Stepper*.

In this manner, the DPUs and LMCs, which possess such Local Controllers, behave as weakly-programmable components, much like the type of weakly-programmable coprocessors used in MpSoCs such as Viper [41] that separate time-critical local control inside the components from non time-critical global control.

3.7.2. AC- Algorithm Controller

This component is a small processor and contains dual-ported local memory. A *send* instruction writes one parameter value via the parameter bus to one particular Local Controller's memory word. And has an optional *wait* mask that identifies one or more Local Controllers' *done* signals, for which the AC should wait before stepping into the next instruction. The *done* signals are used together with the *wait* bitmask to synchronize the operation of multiple

blocks. The *goto* instruction jumps to a particular memory location. This simple instruction set allows the implementation of control loops that depend on the distributed system's state.

The contents of the program memory are described in XML. The parameter bus, AC's wait mask and AC's word width are calculated and synthesized automatically. On Xilinx devices, the AC's program memory uses one or more Block RAMs automatically.

There can be more than one AC per FPGA. Following the compositional FlexWAFE approach, one AC should control one group of inter-related DPUs and LMCs. Global synchronization between groups of non-related DPUs and LMCs is achieved via the inter-block signaling protocol described in section 3.2.

The dual-ported program memory of the AC is reprogrammable at run-time using the Control Bus that is explained in the next subsection.

3.7.3. Control Bus

This bus connects all ACs from all FPGAs to a host PC or to an embedded processor. In the FlexFilm board implementation the datawidth is 16 bit, the address space is 14 bit wide and the I/O FPGA (see fig. 1.3b and fig. 3.2) provides a bridge between the Control Bus and the host PCI-Express (PCIe) bus. This bridge allows the host PC to execute data read and write operations on the control bus. In this implementation, single word read/write accesses take around 4 microseconds (including Operating System (OS) and software overhead), resulting in an average data rate of 500 KByte/second. In other implementations of the FlexWAFE architecture this bridge must be reimplemented in order to allow the used processor to access the control bus. The configuration of the AC (s), parameter bus(es) and local controller(s) and their programs is described in XML file(s) written by the user, and automatically converted into synthesizable VHDL by the FlexWAFE framework. That configuration file(s), and the automated conversion steps will be explained in section 5.2 on page 56.

3.8. Summary and Conclusion

A bottom-up presentation of the building blocks of the FlexWAFE framework was given here. It started by presenting the inter-block communication paradigm followed by the basic blocks: data processing units (DPUs), address generators with optional local reordering caches (LMCs) and distributed local dataflow controllers (ACs). Small examples of how these can be combined with each other and extended with extra functionality were given.

After that, optimized communication with external, off-chip large SDRAMs was discussed. It was shown how the developed controller can prioritize requests, serve multiple clients and maximize the available SDRAM data rates with various scheduling techniques.

The inter-FPGA communication supporting multiple virtual channels with different data rates and zero-time-overhead was explained next. This dedicated solution provides very good performance while minimizing the resource usage due to its static scheduling nature that closely matches the requirements of the applications targeted by this architecture.

The hierarchical control architecture was described in detail, it provides decoupled, distributed control over the multiple data-paths of the system, minimizing control latency on the low levels, and providing just enough flexibility to program stream oriented algorithms.

4. Configuration and Programming

In the previous chapter the FlexWAFE was presented in a bottom-up manner, this chapter presents the pre-synthesis configuration and programming of the architecture. It describes the (synthesis-time) configurable and the (run-time) programmable parts.

To achieve the desired performance, but keep some flexibility, part of the system needs to be specialized and fixed at synthesis-time. Some other parts can be flexible at run-time without sacrificing performance. We therefore made some architectural choices to balance performance and flexibility. The two types of configurability/programmability will be explained in the following sections.

The hardware DPUs were described in configurable VHDL code. Their configuration and interconnections can be changed before synthesizing the FPGA. Some details on how this was achieved were given in the previous chapter and some more details will follow in section 4.1. The instruction set and configuration parameter words for the hardware DPUs and LMCs are configurable via a mix of XML and VHDL code and are explained in the same section.

The program sequences run by the LCs, the result of dynamically using the instruction set defined above, are a key aspect of the FlexWAFE concept. The configurable functions are limited, hence weakly-programmable, nevertheless the ability to change the functionality of the building blocks at run-time is an advantage over fixed architectures. It allows hardware reuse in multiple applications, and to change the behavior of an application at run-time. This will be further explained in section 4.2. Section 4.3 on page 52 summarizes our contributions to the image processing community.

4.1. Hardware Configuration

The signal processing hardware blocks were described in VHDL, one of the two industry standard languages used to describe hardware. The other is Verilog. The code has extensive use of *generics*, *configurations* and *VHDL architectures*, these are language constructs provide a mechanism to describe how the hardware structures depend on certain parameters. This makes the code more generic, hence their name. Several examples on how and why these *generics* were use were given on chapter 3. VHDL *configurations* were used to switch between several sets of *generics* and *VHDL architectures*, allowing one to select a set of generics/architectures for hardware simulation, another one specific for a particular development board, or another for a particular FPGA family all without changing a single line of VHDL code.

To change parameters or operating modes at run-time the architecture provides two types of bus: the control bus and the parameter bus. Both are simple memory mapped parallel buses.

4.1.1. Control bus

This inter-chip bus connects the main CPU (it can be the CPU of a PC, or the CPU of an embedded system) to one or more FPGAs configured with FlexWAFE. The software running on the CPU (master) can change the run-time parameters of the FPGAs (slaves) using this simple bus. The control bus consists of one unidirectional parallel *address bus*, one bidirectional (tri-state) parallel *data bus*, one or more *chip-select* signals (one for each slave FPGA) and one $write/\overline{read}$ signal. The address bus is unidirectional because only the master can do requests on the bus. The data bus is bidirectional because the CPU can perform read and write operations on all FPGAs. This simple bus topology was chosen for its simplicity and availability in almost every single existing CPU. The typical data rate requirement on this bus is very low and some sort of serial protocol would also be possible, but the FPGAs we used had no shortage of free pins and so we decided to use the simpler parallel solution. Due to the use of one exclusive chip-select signal per FPGA, each FPGA has the entire available address space. Every FlexWAFE FPGA can contain one or more ACs (see Section 3.7.2) and each AC occupies one contiguous address region of the control bus. The address regions of ACs from the same FPGA do not overlap, but the address regions of ACs from different FPGAs can overlap because the chip-select signals provide disambiguation. To simplify address decoding and bus arbitration the start of the AC address regions and their size are a power of two. To write to one slave, the master sets the address, data and $write/\overline{read}$ signals and then activates the respective chip-select signal for the duration of one clock cycle. To read from one slave, the master sets the address and $write/\overline{read}$ signals and then activates respective chip-select signal for the duration of one clock cycle. It result of the read will be available on the data bus on the following cycle. One example of a possible bus is presented on Figure 4.1.

The control bus is physically mapped to Printed Circuit Board (PCB) connections and typically interconnects several FPGAs. For these reasons the parameters of the bus can not change often. So we decided to make the address width, data width, number of chip-select signals and ACs address mapping only configurable via VHDL generics. The software code has to be manually kept in sync with these generics. No automation mechanism has been provided by FlexWAFE.

The FlexFilm implementation of the control bus has an address width of 14 bits, an data width of 16 bits and 3 chip-select signals for the 3 FlexWAFE FPGAs. The noise reduction application uses two ACs on the first FlexWAFE FPGA, and one AC on the second, and one AC on the third FlexWAFE FPGA.

The configuration of the ACs is more flexible than that of the control bus, it is done via one XML file per AC. The FlexWAFE framework uses these XML files to generate VHDL files for FPGA synthesis and .h (header) files for software compilation. This way, the framework automatically keeps the hardware configuration consistent with the software configuration. The ACs have four configurable parameter widths: instruction, client wait flags, address and data. These four words form one AC instruction word as depicted in Figure 4.2.

The instruction set of the ACs consists of two instructions: *send* and *goto*. The *send* instruction sends the *parameter data* field to the *parameter address* of the parameter bus. After that it waits for the *done* signals from the LC slaves set in the *client wait flags* to become active be-

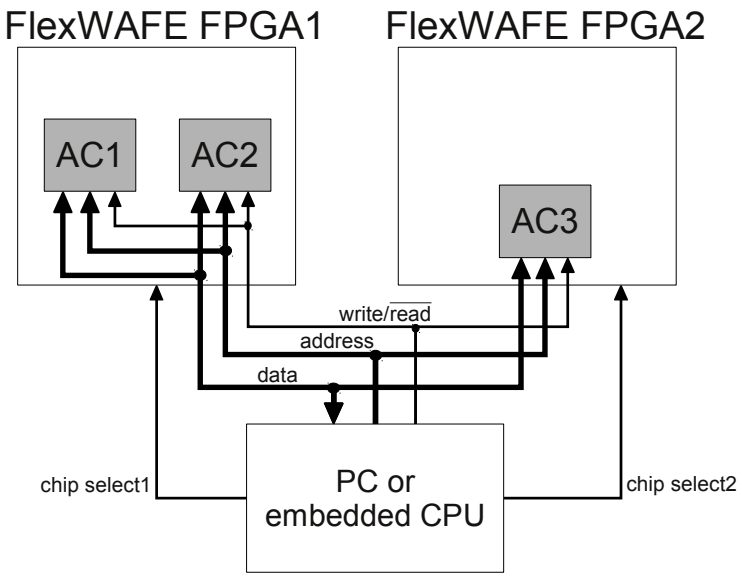


Figure 4.1.: Control bus example with two FPGAs, one contains two ACs, the other one just one.

opcode	client wait flags	parameter address	parameter data

Figure 4.2.: AC instruction word, MSB on the left, LSB on the right, typically 32 bit wide

fore stepping to the next instruction. The *goto* instruction makes the AC program counter jump to the word address defined by the concatenation of the *client wait flags*, *parameter address* and *parameter data* fields. Having just two instructions makes the control logic of the ACs extremely simple (the *instruction* field is a single bit sometimes referred to as `goto_flag`). The control logic basically consists of the program counter - a incrementing counter loadable via *goto* instruction and stoppable via *client wait flags*. This simplicity makes the ACs small and fast, and that was a FlexWAFE requirement.

On the noise reduction application, all ACs had a instruction width of 32 bits with one opcode bit and 16 parameter bits. The width on the wait flags and parameter address was different for each of the four NR ACs. Because the instruction width (32) was twice the width of the control bus data (16), two control bus write commands where needed per instruction word. The FlexWAFE framework automatically detects that and breaks each instruction in the XML file into two write commands. The PC host software writes the four ACs before releasing the reset of the FlexWAFE FPGAs. This ensures consistency of each instruction word, and of each AC program.

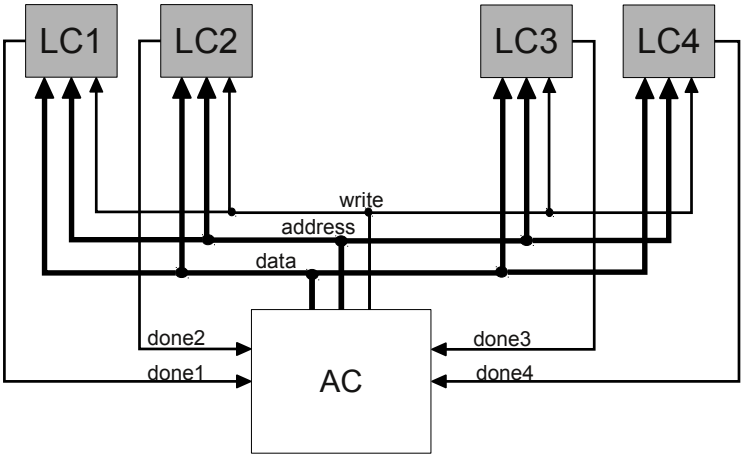


Figure 4.3.: Parameter bus example with four LCs.

4.1.2. Parameter bus

This intra-chip bus connects one AC (master) to one or more Local Controllers (LC) (slaves). If an FPGA contains more than one AC then it shall contain more than one parameter bus, one per AC. This bus consists of one unidirectional parallel *address bus*, one unidirectional parallel *data bus*, one *write* signal, and several *done* signals (at most one per LC). The address space is divided in non-overlapping regions, each LC (see Section 3.7.1) is mapped to a single region. To simplify address decoding the start of the LC address regions and their size are a power of two. The master can only write to the slaves, it can not read from them. A write operation consists in setting the *address* and *data*, then activating the *write* signal for a clock cycle. The slaves can communicate that they have finished their assigned task to the master via the *done* signals. This is a very simple bus designed to use as few hardware resources as possible. One example of a possible bus is presented on Figure 4.3.

The parameter bus configuration is directly derived from the instruction word structure from Figure 4.2. The address bus bit-width is the *parameter address* bit-width. The data bus bit-width is the *parameter data* bit-width. The number of point to point *done* signals is the *client wait flags* bit-width. The address mapping of the several slave LCs is also defined in the AC XML file. This way the code is always consistent with the configuration and address mapping of the parameter bus LCs. The FlexWAFE framework uses all this information to automatically configure the VHDL generics responsible for the parameter bus and inclusive the address mapping of each LC slave.

4.1.3. Local controller

The local controllers have been introduced in Section 3.7.1. They are responsible for delivering the parameters transported by the parameter bus to the LMCs and DPUs. The LCs (masters) are tightly coupled with the LMCs and DPUs (slaves), they provide shadow-register

sets (also called memory sets, see example on Figure 4.4) that are point-to-point connected to the logic of the slaves. Each LC controls a single slave. This massive point-to-point paral-

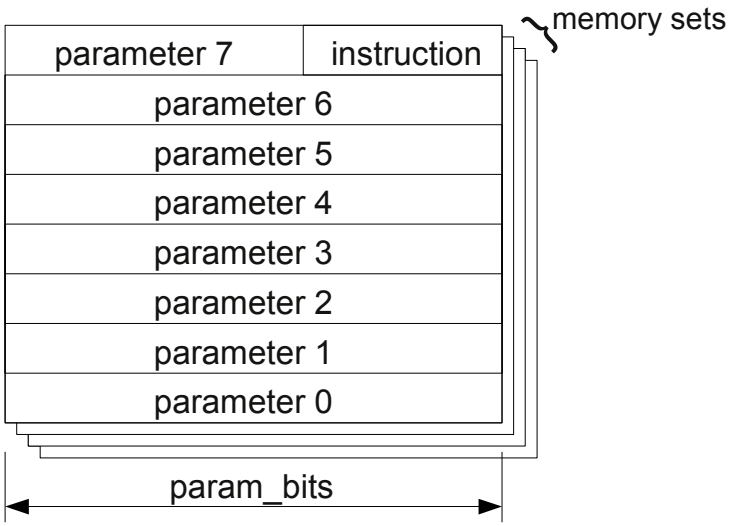


Figure 4.4.: Example of four memory sets of eight parameters each, the instruction field is explained on Figure 4.5

lel connection between the shadow-register sets and the slaves allows single-clock parameter (context) switches. The slaves can communicate that they have finished their assigned context to their LC via the *done* signal, the LC will then switch to the next memory set. If the next memory set contains a reset instruction, the LC switches to the first memory set and issues a *done* pulse to its upstream AC (on the parameter bus side). This is a very simple bus designed to use as few hardware resources as possible. One example of a possible LC to LMC connection is presented on Figure Figure 3.15 on page 43.

The LCs also run their own programs. The operation codes (opcode) of the instruction words are configurable in the AC XML file. A side effect of that is that all LCs connected to a particular AC will have the same opcode mapping. Currently the LCs have the following instruction set: *stop*, *run*, *load* and *reset*. To simplify instruction decoding we decided to dedicate three bits to the LCs instruction word. This also allows to execute two instructions at the same time, for example *reset* and *load*. The typical opcode configuration of the LCs is presented on Figure 4.5

Again the instruction set was kept small to make the LCs small and fast.

4.2. Run-Time Control

Section 4.1 presented how the FlexWAFE system can be configured, this section explains how the control structures operate at run-time.

opcode			Instruction
Bit2 (reset bit)	Bit1 (load bit)	Bit0 (run bit)	
0	0	0	stop
0	1	1	run
0	1	0	load
1	0	0	reset

Figure 4.5.: LC instruction structure

During system initialization the control bus is used to transfer the AC (s) instruction words. The software running on the PC or embedded software first downloads the FPGA (s) bitstreams. After that it programs one AC after the other. Once all ACs have their programs, the FlexWAFE FPGA reset is released and they start operating all at the same time. Each AC then starts to program its subordinated LCs via the parameter bus. The AC programs are structured so that firstly all used LC memory-sets are programed with default reasonable parameters, but are not set to run. At this point the LMCs and DPUs get useful configuration/parameters from memory-set 0 of their LC. After all LC memory sets have been programmed, the ACs set the *run-bit* of memory-set 0 of the LCs that should run. Only at this point in time the LMCs and DPUs start to do something.

Once the LMCs and DPUs are finished with their task, they signal their LC via the *done* signal. The LC then switches to the next memory-set, this only takes one clock cycle, and up-to 16 (in the Xilinx Virtex-II PRO implementation) shadow memory-sets are available. This process repeats itself until the LC finds a reset instruction. Once it does, it signals the upstream AC via its own *done* signal and switches to memory-set 0.

The AC keeps on programing LCs until it finds some active *client wait flags* in its instruction word. Once it does, it waits until all LCs masked in the flags have sent a *done* signal. The *done* signal is a single clock pulse, so the AC has to keep state information about which clients (LC slaves) have already sent a *done* and which didn't. Once all slaves masked in the *client wait flags* have sent pulses, the AC steps to the next instruction. If the next instruction is a *goto* instruction, it jumps to the instruction which address's is the concatenation of the *client wait flags*, *parameter address* and *parameter data* fields (see Figure 4.2).

This very limited, but very lightweight control architecture allows building simple circular memory buffers or to control a complex noise reduction algorithm. This will be demonstrated in the next chapters of this thesis.

4.3. Our contributions

The innovations introduced by FlexWAFE can be resumed to :

- Code reuse by instantiating the same block multiple times in an application. Each instantiation might differ in static parameters (VHDL generics) and in dynamic parameters

(FlexWAFE weak-programmability). This allows high level programmability instead of FPGA dynamic bitstream reconfiguration.

- Code reuse by instantiating the same block in multiple applications. Again due to the flexible nature of the developed blocks, we avoid having to use high-level synthesis of new blocks whenever possible and reuse existing blocks instead.

Both points are orthogonal and can therefore be combined. That has been proved by several student projects [125, 130, 17, 93, 64, 122] where students were given small tasks that they successfully completed by reusing existing blocks and adding new blocks only when necessary. That, together with the automation provided by the FlexWAFE framework described in chapter 5, allowed them to quickly produce the desired results.

5. Design Process and Programming

The previous chapters introduced the FlexWAFE architectural blocks and how they are configured and programmed. In this chapter section 5.1 will explain the design workflow that uses those blocks and section 5.2 the configuring/programming language that allows describing/controlling them. Simple examples will be presented in section 5.3 that will allow the hardware/software co-designer to better understand how to use the FlexWAFE. Section 5.4 will give an overview of the different hardware platforms used to test and validate this design flow. A chapter summary is presented in section 5.5.

5.1. Design Process

As with any other design flow, the FlexWAFE design flow starts by collecting requirements and choosing an appropriate set of algorithms that meet those requirements. A *hardware friendly* implementation of each one of the algorithms is then selected. Those implementations must take into account that FPGAs are massively parallel processing engines and therefore long sequential data dependencies are to be avoided as explained in chapter 2. Data rates and the amount of data storage space should be annotated.

Now each of the algorithms must be split into smaller parts and those should be mapped to the available DPUs and LMCs. To do so, the implementations must in some situations be changed in order to operate on streams and to minimize memory usage by taking advantage of data locality. This can be achieved by using SDF theory and by latter mapping the data processing nodes into DPU blocks and data reordering nodes into LMCs. If one of the required processing blocks does not yet exist in the DPU library, it must be coded in VHDL and added to the library for future use. Tests with the current LMCs have shown that the existing LMCs have enough flexibility to cover the needs of most algorithms and, therefore, only need to be parameterized at synthesis time and (re-)configured by weak-programmability at run-time.

In the second step, all projected blocks are instantiated and interconnected, adding the required Inter-FPGA mux and demux communication blocks, CMC (s) and AC(s). This needs to take into consideration the amount of logic resources available in each FPGA (in the case multiple FPGAs are used), the amount of external memory available per FPGA, and the interconnect data rate between FPGAs.

The third step is to describe parameters that require run-time flexibility in the very simple XML file format described in section 5.2. Each AC requires such a description file. This makes it easier to reuse parts of the code, as opposed to a monolithic solution where a single file for the entire system would be used. Next, the FlexWAFE tool-chain analyzes this XML file and generates the appropriate distributed parameter memories inside each Local Controller, the

microcode for the AC(s), and the synthesis-time parameters for all the blocks. Some VHDL files are also automatically parsed and a suitable `.ucf` (User Constraints File) is generated. If required, physical placement information of performance critical components can be added to the `.ucf` file or to the VHDL source code. At this point, the system can be synthesized and downloaded to the FPGA(s), and the tool-chain will then program the run-time configurable system parameters using once more the information from previous steps.

The parameters defined in step three can be changed if necessary without requiring a new synthesis, as long as they accept the limits defined in other stages (*min* and *max* attributes of the variables section of the XML file). To do so the main processor can write to the control bus, the AC(s) will propagate the change to the parameter bus and the local controller(s) will then provide the new variable value to a DPU or a LMC as detailed in Figure Figure 3.12 on page 39. This hierarchy automatically takes care of synchronization, making sure that all variables change their value at the start of a new data/image stream. At this point the system behavior can be simulated with Modelsim, and the result can be compared with a Matlab or C implementation, if such implementation exists. The FlexWAFE framework contains matlab scripts that partially automate both the simulation process and the comparison process. In the case that a single pixel differs in one image of an image sequence, the script detects it and identifies which pixel at which image differs.

The last step is to write a software program for the host PC or embedded processor (depending on the FPGA board used) to feed data into the FlexWAFE FPGA(s), control the weakly programmable elements in them and to receive the resulting streams. To facilitate this process, a C++ FlexWAFE software library was written that provides a simple Application Programming Interface (API) that encapsulates and abstracts most common operations. In most situations only a couple (ten to one-hundred) lines of code need to be written in this step, because the library takes care of most operations. There are also functions in the software library that allow comparison between expected resulting images and actual resulting images simplifying application debug in situations where the expected results are known beforehand.

A practical example of the design flow as well of the control hierarchy will be presented in Section 6.2.

5.2. Programing using XML descriptions

The programs of the weakly programmable engines of the FlexWAFE architecture are described in Extensible Markup Language (XML) format. XML is an industry standard meta-language widely used to describe data structures in a machine friendly way. Yet it is text based and therefore also human readable. For that reason it has chosen to describe the address space, parameter sets, instructions, constants, etc. of the FlexWAFE. These files describe the programs that the LMCs and DPUs will execute and also describe parts of the hardware that will control their execution (the local controllers and the AC that controls them).

5.2.1. Parameter bus address space

The first section of the file describes the names and addresses of the different LMC and DPU parameters. The addresses are used in the parameter bus that connects the described AC with its subordinated local controllers. Addresses are composed of one or more offset XML elements and one final address XML element, each one has a name attribute. Each LMC or DPU has a name that is formed by concatenating several *name* attributes from the *offset* elements. Each LMC or DPU has its own local controller with the same name as its subordinated LMC or DPU. This local controller will execute instructions that are written in the parameter with *instruction* name attribute and serve the other parameters described in the *final* elements to its attached LMC or DPU. Each parameter name is formed by concatenation of several *names*, and its address by the addition of the several *offsets* with the final address. The *final* elements contain a *range* attribute, that is used to specify how many different values can this parameter take, and therefore how many consecutive addresses it should occupy on the parameter bus. i.e:

```
<address>
   <offset name="lmc_s2c_image_in" base="256">
      <offset name="egress" base="0">
         <final name="offset_addr_step"   range="16">48</final>
         <final name="offset_limit_start" range="16">32</final>
         <final name="instruction"        range="16">0</final>
      </offset>
   </offset>
<address>
```

It describes two parameters one is called *lmc_s2c_image_in/egress/offset_addr_step* with address 304 to 319 and the other *lmc_s2c_image_in/egress/offset_limit_start* with address 288 to 303. The third parameter is actually an instruction to the local controller of the *lmc_s2c_image_in/egress* unit. The *range* attribute has the same meaning as for all other parameters, in this example the local controller will have 16 instructions mapped at the addresses 256 to 271.

So once again, the task of the local controllers is to change some parameters of the LMC or DPU that it is controlling. Typically the number of these parameters (end nodes of type *final*) is a power of two, this ensures a good usage of the available address space. All of the parameters belonging to a local controller need to have the same *range* attribute. This is because the parameters are grouped in *memory sets*, in the example above there are 16 memory sets. Memory set *i* is composed of offset_addr_step[i], offset_limit_start[i] and instruction[i] where *i* is a natural in the interval 0 .. range-1.

5.2.2. Parameter bus address implementation

After the declaration of the parameter names and addresses, the parameter bus that interconnects the algorithm controller and the several local controllers is described:

```
<settings>
   <client_wait_lines>4</client_wait_lines>
   <param_bits>16</param_bits>
   <addr_bits>11</addr_bits>
</settings>
```

The *client_wait_lines* element describes how many local controllers can signal the AC that they completed an assigned task. In most applications, only a small subset of the local controllers needs to do this. Usually the AC works asynchronously to most local controllers. For some applications it needs to synchronize itself with some of the local controllers, and this element declares how many of them are required. The *param_bits* element describes the maximum width in bits of each parameter. The *addr_bits* element is the address with of the parameter bus that interconnects the AC to the local controllers, it must therefore be wide enough to map all the parameters described in the *address* section described above.

5.2.3. Run-time constant parameters

After that some constants are described. These are fixed at synthesis and can not change dynamically at run-time:

```
<constants>
   <const name="maxWidth">4096</const>   <!-- maximum image width -->
   <!-- back_valid lines -->
   <const name="lmc_img_in">1</const>    <!-- bit i_back_valid(0) -->
   <const name="lmc_ref_a">2</const>     <!-- bit i_back_valid(1) -->
   <!-- instructions (hardcoded in the lrag entity) -->
   <const name="stop">0</const>
   <const name="run">3</const>           <!-- run activates 2bits   -->
   <const name="load">2</const>          <!-- bit1 when not running -->
   <const name="reset">4</const>         <!-- bit2 restart from ms=0-->
   <const name="global_base_step">8</const><!-- multiply the extra param -->
</constants>
```

Constants can be used in all parameters as explained in the following paragraph. The user can add as many as necessary, in the example above the first three are user defined constants: the *maxWidth* will be used for example to calculate address patterns, the *lmc_img_in* and *lmc_ref_a* define which local controller is connected to which wait_line input of the AC. The remainder constants are hard-coded in all local controllers and should not be changed nor removed: stop, run, load, reset. These are the opcodes of the instructions that all the local controllers can execute:

stop stops the local controller.

run the local controller will enable this memory set and wait for the completion of its attached LMC or DPU. After that it will step into the next memory set.

load the local controller will send this memory set to its attached LMC or DPU but will not enable it to run.

reset once this memory set is completed (the LMC or DPU signals via a *done* signal that it completed the task) the LC goes to memory set 0.

The opcode of the *stop* instruction is zero. This was chosen because after a FPGA reset the memory that the local controller uses contains zeros. This effectively stops the local controller after a FPGA or power-up reset. The side effect is that the first instruction can only be written (changed from the zero default) after the rest of the program for the local controller has been written. The program counter of each local controller also resets to zero, so that is why the first instruction gets executed after a reset. The opcode of the run instruction has two active bits, bit1 and bit0. These are responsible for sending a new set of parameters to the attached LMC or DPU and activating the LMC or DPU respectively. These opcodes only need 3 bits, but as seen in the example above the *param_bits* are typically 16 or more. As a result, the instruction word uses as many address bits of the bus as the other parameters (remember there are *range* memory sets) but less data bits. In order to avoid this waste of resources, it is possible to use the unused Most Significant Bits (MSB) to carry an extra parameter. The *global_base_step* is a constant (in this example 8) that allows the instruction word to contain such a parameter. In this example the parameter is concatenated with the instruction word in the same addresses. Basically bits *param_bits*-1 down to 3 contain the parameter, bits 2 down to 0 contain the instruction. The parameter needs to be multiplied by the *global_base_step* constant, before being written to the instruction word, and divided by it before being read out of the instruction word. Both operations have zero cost, the first is done in software by the host PC and the second is done in the FPGA with simple wiring because the constant is a power of two therefore the division is a simple fixed bit shift. If in the future the local controllers are expanded to execute more operations, more opcodes are needed. If more opcode bits are used, this constant must be updated. This scheme allows optimized use of the available hardware resources at the expense of a small increase of programming complexity as will be seen in the next paragraphs. Another issue is that the width of this parameter is not *param_bits* like the other parameters in the memory set, but *param_bits*-3.

5.2.4. Run-time variable parameters

The next section contains variables that can be changed at run time.

```
<variables>
   <variable name="width" max="maxWidth">48</variable>
   <variable name="height" max="maxHeight">48</variable>
</variables>
```

As with the constants section the number of variables is not limited and the user can define as many as necessary. The variables have a *name* attribute and a *max* attribute. This is used by the software to validate that the value of the variable is between 0 and *max*. The synthesis tool also uses the *max* value to calculate the resources required to store the parameters. The *max* value can be a number, or a previously defined constant like on the example above.

5.2.5. Control program

The final section contains the values of the parameters defined in the first section.

```
<code>
  <send dest="me_engine/input_control/h_blocks">width/blockWidth-1</send>
  <send dest="lmc_img_in/egress/instruction" ms="1">
                                          2*global_base_step+stop</send>
  <send dest="lmc_apt/egress/offset_base" ms="0">
                                            max(blockWidth,ppb)-1</send>
  <send dest="dpu_g/filter" ms="0" label="filt">1</send>
  <send dest="dpu_g/filter" ms="2" waitfor="lmc_ref_a">4</send>
  <goto>filt</goto>
</code>
```

The first line of the example above calculates the result of the expression *width/blockWidth-1* where width and blockWidth are previously defined constants or variables. The expression can have the four arithmetic operations addition, subtraction, multiplication and division with the normal precedence rules. Parenthesis, and min(), max(), sqrt() and log() functions are also supported. The destination parameter is coded in the *dest* attribute, and in the optional *ms* (memory set) attribute. When *ms* is not specified like on the first line of the example above, it defaults to zero. The second line shows an example of setting the value 2 for the *global_base_step* parameter and stop the instruction of memory set 1 of the *lmc_img_in/egress* local controller. The third line has an example on how to use the *max()* function. Other functions have similar syntax, arguments are separated by comas.

The send instructions are executed sequentially by the AC (s). It sends one instruction after the other via the parameter bus until it finds a *send* element with a *waitfor* attribute. Then it waits for the completion signal from the LMCs and/or DPUs described in that attribute. The synchronization is like a barrier that only completes when all the completion signals have been active for at least one cycle. There is no need for the signals to be active at the same time. The send instructions can also have a *label* attribute. It is used as a destination of the *goto* instructions, that make the execution (instruction pointer) of the AC jump to the location of the label.

5.2.6. Framework for work-flow automation

The user-written software application invokes functions from the FlexFilm device driver to upload the programs to the several FPGAs of the FlexFilm board. The device driver first validates the user supplied XML description files, then converts them into a format that is up-loadable to the ACs that they are designated to. The programs are uploaded to the I/O FPGA via PCIe where a bridge forwards them via the control bus to their destination FPGA. There can be multiple ACs per FPGA, and each one can have a different program. In order to distinguish them, each FPGA has a separated address space on the control bus and each AC occupies a segment of the FlexWAFE FPGA's 14-bit address space. The device driver program upload functions take therefore 3 parameters: XML filename, destination FPGA (0, 1 or 2) and destination base address within the 14-bit address space of the targeted FPGA.

This is achieved by a program invoked from a makefile and it automatically also generates a .xco (Xilinx pre-compiled object file) file for synthesis of the AC and respective parameter bus and local controllers.

5.3. Program examples

One of the tasks that is often required by image processing algorithms is to create and manage a circular buffer containing two or more image frames. An introduction to circular buffering can be found in Appendix B. This task is easily achieved using the FlexWAFE framework. When the amount of memory per buffer is small, a LMC_APT (see Section 3.4.4) should be used, otherwise a LMC_S2C combined with LMC_C2S are a better choice. These last two access the external SDRAM memory via the CMC described in section 3.5 and are synchronized via the parameter bus by an algorithm controller. An example application of image frame double-buffering will be described in the next paragraphs. Two buffers were chosen for simplicity reasons, but the system can easily by extended to n buffers. First the address space of the registers in the parameter bus must be described:

```
<address>
   <offset name="image_in" base="0">
      <offset name="egress" base="0">
         <final name="instruction"         range="16">112</final>
         <final name="offset_base_start"   range="16">96</final>
         <final name="offset_base_step"    range="16">80</final>
         <final name="offset_base_end"     range="16">64</final>
         <final name="offset_addr_step"    range="16">48</final>
         <final name="offset_limit_start" range="16">32</final>
         <final name="global_base_start"   range="16">16</final>
         <final name="global_base_end"     range="16">0</final>
      </offset>
   </offset>
   <offset name="image_out" base="128">
      <offset name="ingress" base="0">
         <final name="instruction"         range="16">112</final>
         <final name="offset_base_start"   range="16">96</final>
         <final name="offset_base_step"    range="16">80</final>
         <final name="offset_base_end"     range="16">64</final>
         <final name="offset_addr_step"    range="16">48</final>
         <final name="offset_limit_start" range="16">32</final>
         <final name="global_base_start"   range="16">16</final>
         <final name="global_base_end"     range="16">0</final>
      </offset>
   </offset>
<address>
```

The *image_in* instance is a LMC_S2C and inputs the images into memory, the *image_out* instance is a LMC_C2S and reads the images out of the memory. Now the parameter bus must be configured:

```
<settings>
   <client_wait_lines>2</client_wait_lines>
   <param_bits>16</param_bits>
   <addr_bits>11</addr_bits>
</settings>
```

Then some constants are defined:

```
<constants>
  <const name="maxWidth">4096</const>  <!-- maximum image width in pixels -->
  <const name="maxHeight">4096</const> <!-- maximum image height in pixels -->
  <const name="ppb">16</const><!--(p)ixels (p)er SDRAM memory access (b)urst-->
  <!-- back_valid lines -->
  <const name="image_in">1</const>  <!-- bit i_back_valid(0) from pb_server -->
  <const name="image_out">2</const> <!-- bit i_back_valid(1) from pb_server -->
  <!-- instructions (hardcoded in the lrag entity) -->
  <const name="stop">0</const><!-- all bits zero -->
  <const name="run">3</const><!-- run activates bit0(runbit) and 1(loadbit)-->
  <const name="load">2</const><!-- bit1 (loadbit) is used when not running -->
  <!-- bit2 (resetbit) to restart from ms=0 when finished-->
  <const name="reset">4</const>
  <!-- multiply the global base step whith this-->
  <const name="global_base_step">8</const>
</constants>
```

In order to be able to change the size of the image buffers dynamically variables must be declared for latter use:

```
<variables>
   <!--default image width [pixels]-->
   <variable name="width" max="maxWidth">48</variable>
   <!--default image height [pixels]-->
   <variable name="height" max="maxHeight">48</variable>
</variables>
```

Finally the program that the AC will execute to control the two LMCs is defined. It starts by programing all LMC parameters:

```
<code>
   <!-- input to the first image buffer -->
   <send dest="image_in/egress/offset_base_start"  ms="0">0</send>
   <send dest="image_in/egress/offset_base_step"   ms="0">maxWidth/ppb</send>
   <send dest="image_in/egress/offset_base_end"    ms="0">maxWidth - maxWidth/ppb</send>
   <send dest="image_in/egress/offset_addr_step"   ms="0">1</send>
   <send dest="image_in/egress/offset_limit_start" ms="0">width/ppb - 1</send>
   <send dest="image_in/egress/global_base_start"  ms="0">maxHeight/ppb*0</send>
   <send dest="image_in/egress/global_base_end" ms="0">maxHeight/ppb*0 + height/ppb - 1</send>
   <send dest="image_in/egress/instruction"        ms="0">1*global_base_step+load</send>
   <!-- input to the second image buffer and go back to the first buffer -->
   <send dest="image_in/egress/offset_base_start"  ms="1">0</send>
   <send dest="image_in/egress/offset_base_step"   ms="1">maxWidth/ppb</send>
   <send dest="image_in/egress/offset_base_end"    ms="1">maxWidth - maxWidth/ppb</send>
   <send dest="image_in/egress/offset_addr_step"   ms="1">1</send>
   <send dest="image_in/egress/offset_limit_start" ms="1">width/ppb - 1</send>
```

```
   <send dest="image_in/egress/global_base_start"  ms="1">maxHeight/ppb*1</send>
   <send dest="image_in/egress/global_base_end" ms="1">maxHeight/ppb*1 + height/ppb - 1</send>
   <send dest="image_in/egress/instruction"        ms="1">1*global_base_step+reset+load</send>

   <!--  output from the first image buffer -->
   <send dest="image_out/ingress/offset_base_start"  ms="0">0</send>
   <send dest="image_out/ingress/offset_base_step"   ms="0">1</send>
   <send dest="image_out/ingress/offset_base_end"    ms="0">width/ppb -1</send>
   <send dest="image_out/ingress/offset_addr_step"   ms="0">maxWidth/ppb</send>
   <send dest="image_out/ingress/offset_limit_start" ms="0">(blockHeight-1)*maxWidth/ppb</send>
   <send dest="image_out/ingress/global_base_start"  ms="0">maxHeight/blockHeight*0</send>
   <send dest="image_out/ingress/global_base_end"    ms="0">
                               maxHeight/blockHeight*0 + height/blockHeight - 1</send>
   <send dest="image_out/ingress/instruction"        ms="0">1*global_base_step+load</send>
   <!--  output from the second image buffer and go back to the first buffer -->
   <send dest="image_out/ingress/offset_base_start"  ms="1">0</send>
   <send dest="image_out/ingress/offset_base_step"   ms="1">1</send>
   <send dest="image_out/ingress/offset_base_end"    ms="1">width/ppb -1</send>
   <send dest="image_out/ingress/offset_addr_step"   ms="1">maxWidth/ppb</send>
   <send dest="image_out/ingress/offset_limit_start" ms="1">(blockHeight-1)*maxWidth/ppb</send>
   <send dest="image_out/ingress/global_base_start"  ms="1">maxHeight/blockHeight*1</send>
   <send dest="image_out/ingress/global_base_end"    ms="1">
                              maxHeight/blockHeight*1 + height/blockHeight - 1</send>
   <send dest="image_out/ingress/instruction"        ms="1">1*global_base_step+reset+load</send>
```

After that step, the local controllers of each LMC will each have its full program loaded. So now the AC task is just to start/stop the two local controllers in order to sync the buffers:

```
   <!-- start inputing the very first image and wait for it to finish -->
   <send dest="image_in/egress/instruction" ms="0" waitfor="image_in">
      1*global_base_step+run</send>

   <!-- input to the second image buffer -->
   <send dest="image_in/egress/instruction" ms="1" label="mylabel">
      1*global_base_step+reset+run</send>
   <send dest="image_in/egress/instruction" ms="0">1*global_base_step+load</send>
   <!-- output from the first image buffer -->
   <send dest="image_out/ingress/instruction" ms="0">1*global_base_step+run</send>
   <send dest="image_out/ingress/instruction" ms="1" waitfor="image_in+image_out">
      1*global_base_step+reset+load</send>

   <!-- input to the first image buffer -->
   <send dest="image_in/egress/instruction" ms="0">1*global_base_step+run</send>
   <send dest="image_in/egress/instruction" ms="1">1*global_base_step+reset+load</send>
   <!-- output from the second image buffer -->
   <send dest="image_out/ingress/instruction" ms="1">1*global_base_step+reset+run</send>
   <send dest="image_out/ingress/instruction" ms="0" waitfor="image_in+image_out">
      1*global_base_step+load</send>

   <goto>mylabel</goto>
</code>
```

Notice that because in this example $ms1$ is that last memory set used on the local controllers, it always has the $+reset$ parameter associated to it. Without it, the local controllers would start serving parameters from memory sets that were not yet programmed.

5.4. Implementation on multiple hardware platforms

The FlexWAFE architecture was designed to be as hardware independent as possible. This makes the task of migrating to a newer FPGA technology or a newer board easier. To achieve

this, the VHDL code avoids using direct instantiation of Xilinx specific FPGA structures and can be fully configured via generics. The automation scripts are also generic and can be reused with little or no changes. The FlexWAFE suite was successfully tested on the following boards:

- **FY7206 board from Thomson** it contains seven XC2V2000 FPGAs, one does the I/O and the other six are connected to this one in a ring-like manner building a processing chain. It was used to test an early version of the inter-FPGA communication blocks and the color space conversion DPUs. An analog I/O hardware was developed for this board by [48]. A digital I/O firmware was developed for this board by [17].
- **Spartan 3 board from Digilent Technologies** contains a single XC3S200 FPGA. It was used to test the discrete wavelet transform DPUs. Due to the small size of the FPGA only one level of 2D direct followed by one level of 2D inverse wavelet transformation were implemented.
- **2VP7 board from Avnet** contains a XC2VP7 FPGA and SDRAM. It was used to test the CMC SDRAM memory controller and some of the LMCs.
- **FlexFilm board from Thomson** contains four XC2VP50 FPGAs and SDRAM.
- **SX240T board from HiTech Global** [57] contains one Virtex 5 SX240T FPGAs and SDRAM. This port was **not** done by the author of this thesis.

Experiments proved that the FlexWAFE architecture and its framework can be easily ported to all these platforms.

However, a large scale case study was conducted on the FlexFilm platform and will be described in the following chapter.

5.5. Summary and Conclusion

This chapter presented the design workflow used by the FlexWAFE architecture and framework. It started by presenting step by step instructions on how to combine the existing FlexWAFE blocks to implement a set of algorithms. Followed by an explanation of the tasks that the framework automates in order to ease and speed-up the design process.

After that, a detailed explanation of the *xml* based system description together with detailed information of the control blocks, AC and local controller were given.

The XML based programing language for the weakly-programmable control structures allows reuse and simplifies the burden of programing such a distributed concurrent system.

Section 5.4 briefly presented the five hardware platforms that were used to test and validate the FlexWAFE architecture and framework.

6. Case Study

This chapter presents the practical side of this thesis, the implementation of multiple algorithms using the framework described in chapter 3. It is dynamically configured like explained on chapter 4 and follows the design and programming work-flow previously described in chapter 5. The chapter starts with a more in-depth description of the main hardware platform used for the development of the architecture, the FlexFilm board, in section 6.1. This is followed by a description of the PC interface firmware and software device drivers developed for efficient communication with the FlexFilm hardware resources.

Section 6.2 will describe the implementation of the film grain noise reduction algorithm that was the main application example of the FlexFilm project and of the FlexWAFE architecture. That section deals mainly with the signal processing aspects of the implementation.

Section 6.3 will describe how the FlexWAFE interacts with the application example above.

This chapter is concluded by a summary and an outlook on possible future development directions in section 6.4.

6.1. Communication with a FlexFilm Board

6.1.1. FlexFilm Hardware

The FlexFilm hardware consists of a PCIe 1.0 expansion board for standard PCs with four XC2VP50 FPGAs. It has been described in detail in Section 1.2.5, is depicted in Figure 1.3a on page 11, and its block diagram can be seen in Figure 1.3b. The following subsections will present a top-down overview on the practical usage of this platform to implement a image processing algorithm.

6.1.1.1. Inter-Board Communication and I/O-FPGA

For inter-board communication (host-to board or board to board for a multi-board setup) a initial bit budgeting was performed. Transferring a single 2K image stream in and out simultaneously requires a data rate of 2 $\times$ 3 Gbit/s = 6 Gbit/s transferring several streams and/or streams at higher resolutions requires even more. For inter-board communication a certain amount of flexibility is required to set up systems consisting of a variable amount of boards in combination with storage devices for content provision and video cards for visual feedback. Several network-based communication standards were evaluated by Heithecker [49] and finally PCIe was selected and implemented by Thomson Grass Valley.

The I/O-FPGA has a fixed configuration which is loaded at power-up from an external flash memory and allows:

- 8 Gbit/s bidirectional PCIe link to the host PC;
- to configure the 3 FlexWAFE FPGAs by the host PC via PCIe in less than a second;
- direct board-to-board communication via a second dedicated 8 Gbit/s bidirectional PCIe link which however currently remains unused.

6.1.1.2. Inter-Chip Communication

For image stream transport between the FPGAs of Figure 1.3, a FPGA resource usage and bit budgeting was performed by Heithecker [49]. As a result, shared bus-type architectures such as Peripheral Component Interconnect (PCI) or PCI-Extended (PCI-X) where not taken into account due to data rate availability, scalability and PCB layout issues. On the other side, network-like communication architectures such as PCIe, HyperTransport (HT) or RapidIO (RIO) including switching devices were considered too complex since only a limited, fixed number of FPGAs need to be connected.

For these reasons, a lightweight point-to-point structure was chosen in which FPGAs are directly coupled using 16 parallel unidirectional LVDS links providing a data rate of 8 Gbit/s. Due to board size, layout and FPGA pin restrictions, a chain structure using six links was implemented by Thomson Grass Valley for the FlexFilm board as seen in Figure 1.3 on page 11.

6.1.1.3. External SDRAM

The total amount of available embedded dedicated RAM per FPGA is just 4,176 Kib[1] or 522 KiB. This is just 4.5 % of the amount required by a 2K-frame with 30 bit per pixel (11.25 MiB or 11520 KiB, refer to Table 1.1 on page 5), and even if a massive amount of slices would be used as RAM (distributed RAM) it would not be enough to hold even a single frame. Therefore, attached to each FPGA are 512 MiB external Double Data Rate SDRAM (DDR-SDRAM), organized as 4 independent 32-bit channels of 128 MiB each. Two channels are located at the left edge of the chip and two on the right edge of the chip[2]. They can be paired together to form two 64-bit channels, one in each side of the FPGA. The channels operate at the same clock frequency as the FPGA, 125 MHz. This gives a total maximum theoretical data rate of 14 Gbit/s per FPGA.

6.1.2. FlexFilm Object Oriented API

The FlexFilm board fits in one PCIe 4x expansion slot of a PC. The host CPU (s) of the PC will then run application software that will control the FlexFilm board, which will be the topic of this subsection.

[1] 1Kib=1024 bits, see section A.1 (page 109) for details

[2] Due to a rotated chip placement the RAMs in Figure 1.3 appear at the top and bottom edge.

The board application field is mostly image processing of high-resolution image sequences. Such sequences tend to occupy large amount of storage space and require high resolution displays to be properly viewed. With the increase of hard disk storage capacity and speed in the last few years, any current PC is able to store such sequences, and recent graphic cards and LCD PC monitors are capable of displaying images of up to 3200x2400 pixels (QUXGA resolution). To make efficient use of these resources the FlexFilm software uses memory caching for high speed streaming of image sequences read from the hard drives, and OpenGL to access the graphic card memory-buffers directly. Furthermore multiple threads were used in order to decouple the DMA reading and the writing of image streams from/to the board. This achieves maximum throughput and avoids stalls and dead locks that could occur due to the non-deterministic scheduling of the PCIe communication by the OS, but mainly due to the usage of a non real-time OS.

To facilitate testing and development, all interface functionality with the FlexFilm board was coded into one object oriented library, applications that use the board only need to link against it and call its methods. The features provided by this application library are enumerated in the following list.

- Separated and independent DMA based image read and write. Although independent they are synchronized with a semaphore to avoid image buffer overflows or underflows. This allows automatic pipelining of simultaneous bidirectional image transfers.
- Blocking read and write operations for simpler synchronization. The API and the application that uses it can therefore be faster and simpler.
- Supports multiple FlexFilm boards in one host PC.
- Image size can change at the start of every image without requiring extra commands to be sent.
- Image sizes can be as small as 8x8 pixels and as big as the application requirements.
- Control bus routines are optimized, and use the host endianness, and convert to the board endianness only if necessary.
- Can read little and big-endian Digital Picture Exchange (DPX) files.

Tests conducted with the FlexFilm platform have shown that faster than real-time image transfers are achieved (37 frames per second (fps) @ 2048x1536 pixels, 10 bits per pixel (bpp)). These tests load around fifty images from disk to main memory and then stream them to the FlexFilm board, where they pass trough all FPGAs and are then streamed back to the main memory to be displayed by the graphic card. We attempted to stream the images directly from hard-disk to the FlexFilm board in order to be able to process longer image sequences. But the hard-disks available to us at the time were not able to provide the required data rate and therefore did not meet out real-time requirements. The results we got with memory transfers are also achievable with longer image sequences if faster (more expensive) hard-drives or dedicated Redundant Array of Independent Disks (RAID) systems are used. Furthermore, PCs have multiple PCIe slots and we equipped one with two FlexFilm boards and redid the tests. Due to the point-to-point nature of PCIe there was no speed penalty and both boards performed at the same speed as a single board.

6.1.3. FlexFilm Device Driver

In order to communicate with specialized hardware like the FlexFilm board, special, hardware-dependent device driver is required. Such a device driver for the Linux operating system was written from scratch in the course of this thesis work. It was written in C programming language using the *Linux Device Drivers* [27] book as reference.

High-speed data transfers to and from the FlexFilm board were done via hardware-assisted transfer of large blocks of contiguous memory, called *huge pages* (2MiB big). To transfer one 2K image in .dpx file format, eight such blocks are required. The device-driver instructs the hardware to start a transfer of such a block and once the transfer is finished the device driver gets notified via an interrupt. This mechanism is commonly known as direct memory access (DMA) and is very lightweight for the main processor because it is free to perform other tasks while the transfer is taking place.

The device driver also provides functions to reprogram each one of the three FlexWAFE FPGAs, read their internal temperature and access the control bus. It provides error handling via timeouts and has multi-board support. Two boards were successfully tested simultaneously.

6.1.4. FlexFilm Firmware

The I/O FPGA in the FlexFilm board contains a fixed configuration, because it is meant to do I/O and manage the other three FPGAs. Therefore, it does not need to change its configuration.

This FPGA was developed together with Thomson Broadcast, one of the project partners in the FlexFilm project. Their contributions were:

- integrate one 4x PCIe 1.0 IP core from Xilinx. This provides communication to the PC motherboard.
- DMA controller. Capable of transferring large blocks of data to and from the motherboard main memory.
- FPGA programming bridge. Allows programing each one of the three FlexWAFE FPGAs with a data stream coming from the PCIe.
- I^2C bus bridge. Allows reading thermal information from the FPGAs.
- Dot display driver. To display status information on the 4 characters dot display in the back of the FlexWAFE board.

Our colleague Sven Heithecker programmed two CMC, one for each SDRAM memory bank of the I/O FPGA and TDM based communication channels for inter-FPGA communication.

During this thesis, an interrupt controller and a bidirectional control-bus bridge were added. The interrupt controller allowed increasing the original image transfer speed from 9 fps to 37 fps. The original image transfer logic provided by Thomson was polling based and not very efficient. The interrupt controller together with an improved software device driver presented in the previous section reduced the transfer overhead which enabled the sustained rate of 37 fps.

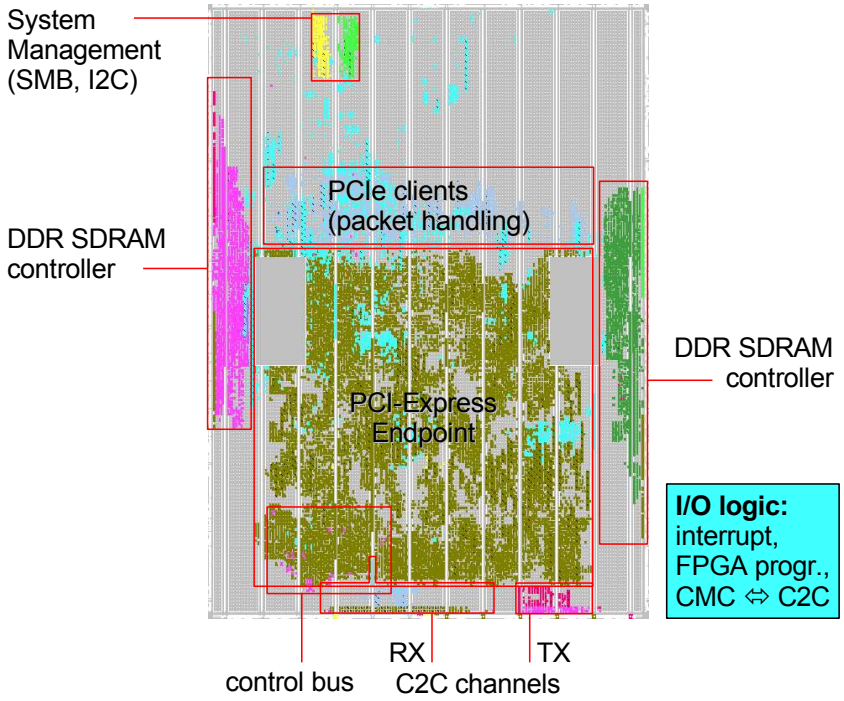


Figure 6.1.: I/O FPGA floorplan

The control-bus bridge provides the means for the host-PC to communicate with the FlexWAFE control infrastructure. The final FPGA floor-plan is presented in Figure 6.1.

6.2. Noise reduction implementation on FlexFilm

The noise reduction application and algorithm were already introduced in Section 1.2.4 on page 9 and can be seen in Figure 6.2. The next sections will detail their implementation starting with the motion estimation, and the motion compensation followed by the discrete wavelet transformations and concluded with their mapping into the three FlexWAFE FPGAs of the FlexFilm board.

6.2.1. Motion Estimation

Motion Estimation (ME) detects object movement between consecutive images. ME is performed in the luminance (Y) color space. Software simulation tests have shown that three independent ME, one in each RGB color channel, would yield better results in the following filtering stages because RGB→Y is not an injective function. However, the increase of resources that those three independent ME would cause is prohibitive and the compromise is to use a single ME in the non-injective Y space and then use the resulting motion vectors for motion compensating all three RGB color components.

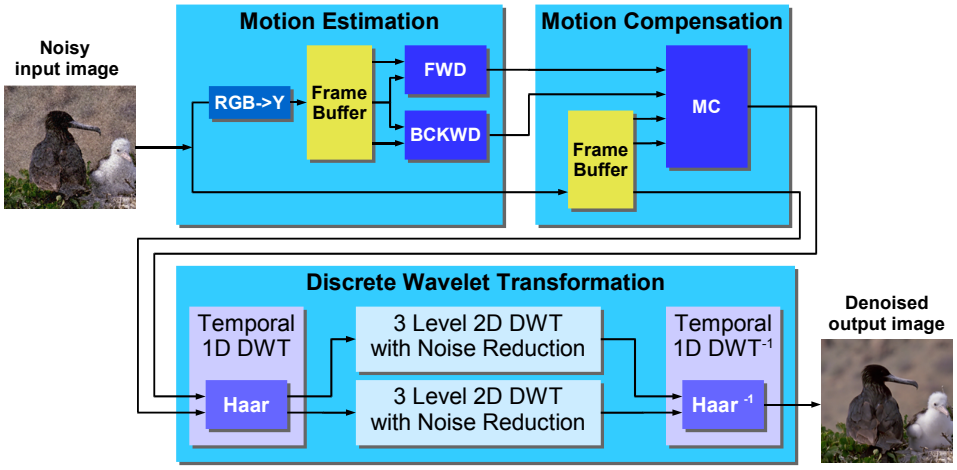


Figure 6.2.: Complete noise and grain reduction algorithm

For ME, the reference and neighboring images are divided into blocks of $M \times N$ pixels, and for each block of the neighboring images the position of the corresponding block in the reference image is searched within a displacement, which is called Motion Vector (MV) of $[-r, +r-1]$ pixels (vertical component) and $[-p, +p-1]$ pixels (horizontal component) as seen in Figure 6.3. Translation image pixel movements within these intervals are correctly detected by this method.

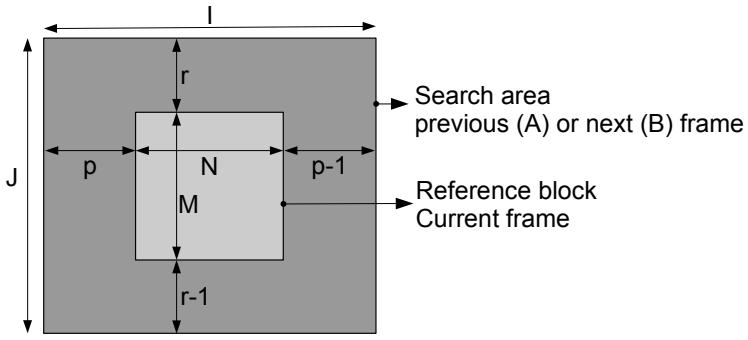


Figure 6.3.: Reference image block and its search area in the previous or next frame

The size of these blocks has a strong impact on the image quality of the motion compensated image (Section 6.2.2 on page 77) . The smaller the block size, the less artifacts will appear between the borders of the motion compensated image blocks. However, the motion estimation algorithm might not be able to find a correct matching block, due to noise of lack of structured features, resulting in incorrect motion vectors and incorrect motion compensated images [58]. On the other hand, large size blocks may produce a less accurate motion vector since a large block may contain two or more objects moving at different speeds and directions [20]. A balance between quality and complexity must be found. Experiments made by [42] show that a block size of 8×8 or 16×16 pixels provides good results and can be efficiently implemented in hardware.

Rotation, resizing and deformation of objects in the image can create pixel movements that are not fully correctly detected using this method. To correctly detect rotation and resizing, the search space would have to contain rotated and re-sized versions of the reference block, which creates a undesirably big search space. To detect object deformation, dense matching (pixel based motion estimation) must be used [78]. Dense matching is more computationally demanding than the method we choose, and would not be implementable using the FPGAs of the FlexFilm board. Image objects occlusions and objects that move outside of the image frame can also cause MV miss-estimation in our method.

Due to image changes, it is likely that the blocks will never be exactly the same. The Sum of Absolute Differences (SAD) of all block pixels is calculated as a matching criteria for each possible MV and the MV with the lowest SAD is used. To achieve bidirectional motion estimation we simply use a motion estimation engine that compares the current image with the previous one, and a second motion estimation engine that compares the current image with the next one.

An exhaustive search algorithm is used [105, 64], which computes the SAD values of all $2\times 2r\times 2q$[3] possible MVs, resulting in a requirement of about 256 Gops/s (subtraction, absolute value, accumulation and comparison at 125MHz). While iterative algorithms exist which reduce the required amount of operations, the advantage of the exhaustive search is its regular structure, which allows an easy implementation and the data-independent memory access and deep memory prefetching (see [49]). A second advantage is that the exhaustive search guarantees optimal results (it finds the global minimum), whereas iterative solutions do not (can get stuck on local minimums). While for compression algorithms a miss-prediction only leads to a reduced compression factor, for the noise reduction algorithm a miss-prediction may lead to erroneous removal of grain and therefore to a loss of quality. Therefore full (exhaustive) search is preferred.

The generalized formula for SAD computation is presented in Equation 6.1. Where img_{ref} are the pixels in the reference image block, img_{search} are the pixels in the image block where the search is performed, M is the image block height, N its width, i is in the interval $[-r, +r-1]$ and j is in the interval $[-p, p-1]$. After calculating $SAD(i,j)$ for all i,j the minimum $SAD(i,j)$ and the corresponding MV (an i,j pair) are taken as the motion estimation result of the image block.

$$SAD(i,j) = \sum_{m=1}^{M}\sum_{n=1}^{N} |img_{ref}(m,n) - img_{search}(m+i, n+j)| \tag{6.1}$$

In our system images are transferred at one pixel per clock cycle, therefore it is desirable to process an image block of $M\times N$ pixels in $M\times N$ clock cycles. Doing so allows processing throughput to match image transfer throughput, effectively avoiding stream stalls. Equation 6.1 can be decomposed into four loops presented in Equation 6.2 and it can easily be seen that the SAD complexity for each image block is $O(2r\times 2p\times M\times N)$.

[3]Our implementation uses 16×16 image blocks within a search area of $(7-(-8)+1)^2$, for previous and succeeding image

```
for (i=-r; i < r; i++) {
  for (j=-p; j < p; j++) {
    for (m=1; m ≤ M; m++) {
      for (n=1; n ≤ N; n++) {
        SAD(i,j) += |img_ref(m,n) − img_search(m+i,n+j)|
      }
    }
  }
}
```
(6.2)

However, as explained before it is highly desirable to reduce that complexity to $O(M \times N)$. In order to do so $2r \times 2p$ independent SAD calculation units are required. Each one of them computes $SAD(i,j) \mathrel{+}= |img_{ref}(m,n) - img_{search}(m+i, n+j)|$ for one particular (i,j) pair; that is, for one of the possible locations of the reference block within the image block search area. In other words: the **i** and **j** loops are done in parallel. Each of these calculation units is called a Processing Element (PE).

Image transfers (and most times image processing) are done row-wise from left to right, top to bottom. Therefore we also want to produce the MVs in the same order, to facilitate the following motion compensation and DWT filtering processing steps. To do so, Equation 6.2 must be reordered into:

```
for (n=1; n ≤ N; n++) {
  for (m=1; m ≤ M; m++) {
    for (j=-p; j < p; j++) {
      for (i=-r; i < r; i++) {
        SAD(i,j) += |img_ref(m,n) − img_search(m+i,n+j)|
      }
    }
  }
}
```
(6.3)

The outer loop now moves from left to right, just like the image input stream usually does, and this produces MVs by that same order. Now that the order of the two outer loops is defined, lets take a look at the data demands of the two innermost loops, the ones that must run in parallel. The required number of pixels are $M \times N$ pixels from the reference image and $2r \times 2p$ pixels from the search image. For any given (m,n) position at the input of the array, $img_{ref}(m,n)$ and $img_{search}(m+i, n+j)$ are required. The first one is easy, all PEs require the very same pixel $img_{ref}(m,n)$ from the reference image. But the second one is harder because every $PE(i,j)$ requires a different pixel $img_{search}(m+i, n+j)$ from the search image. That would lead to huge data-rate and FPGA routing demands, $2r \times 2p + 1$ pixels would have to be fetched in a single clock cycle, and routed independently to each PE.

By setting $r = M$ and $p = N$ the search area will be four times the size of the reference block and neighboring reference image blocks will share half of the search image pixels. The right half of the search area on any image reference block will be the left half of the search

area on the image reference block to it's right. By pipelining the processing elements in a particular way it should be possible to use the above fact to avoid fetching the overlapping search areas twice. The pixels will be fetched once, and used twice, one time for the left half of the current reference image block and at the same time for the right half of the next image reference block. But the required data from the search area will still be twice the one from the reference image. To cope with that we fetch two pixels simultaneously from the search area, and a single pixel from the reference area. Fetching these three pixels per clock cycle will allow us to process every $M \times N$ image reference block in $M \times N$ clock cycles. One of the search data streams will fetch the upper-half of the search area, the other one will fetch the lower-half.

We pipeline the calculations temporally by grouping the PEs to form an unidimensional systolic array. A systolic system is a network of processors which rhythmically compute and pass data through the system. Every processor regularly pumps data in and out, each time performing some short computation, so that a regular flow of data is kept up in the network. These systolic arrays enjoy simple and regular communication paths, and almost all processors used in the networks are identical [72].

The systolic array also allows starting to process the next image block back-to-back with the current one in a gap-less manner when $M = N$ and $r = p = \frac{M}{2}$. The entire image can be processed pixel by pixel and block by block without the need to insert artificial idle cycles.

The Systolic array data requirements have been reduced by the techniques described above, and three data streams have been defined: *reference* (Equation 6.4), *search upper* (Equation 6.5) and *search lower* (Equation 6.6).

```
for (i=0; i < img_height; i+=M)
  for (j=0; j < img_width; j+=N)
    for (n=0; n<N; n++)                                   (6.4)
      for (m=0; m<M; m++)
        reference = img_ref(i + m, j + n)
```

```
for (i=0; i < img_height; i+=M)
  for (j=0; j < img_width; j+=N)
    for (n=-p; n<p; n++)                                  (6.5)
      for (m=(i==0?0:-r); m<r; m++)
        search_upper = img_search(i + m, j + n)
```

```
for (i=0; i < img_height; i+=M)
  for (j=0; j < img_width; j+=N)
    for (n=-p; n<p; n++)                                  (6.6)
      for (m=r; m<(i<img_height-M?3:2)r-1; m++)
        search_lower = img_search(i + m, j + n)
```

Figure 6.4 presents a practical example. The three streams must be read with the correct order and synchronized with each other before being delivered to the array. To achieve this we reused existing code. All images are stored in external SDRAM memory, after being

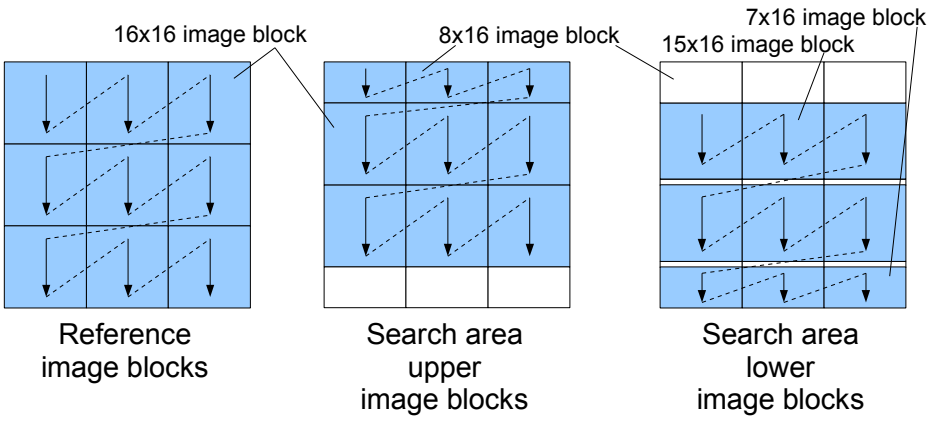


Figure 6.4.: Reference image blocks and its search upper and lower areas in the previous (ME A) or next frame (ME B) for $M = N = 16$, $r = p = 8$ and $img\_height = img\_width = 48$

converted from RGB to Y. To do so an LMC_S2C stores the input image stream using a four image circular buffer in SDRAM as depicted in Figure 6.5. Appendix B details how these buffers are built and how they work.

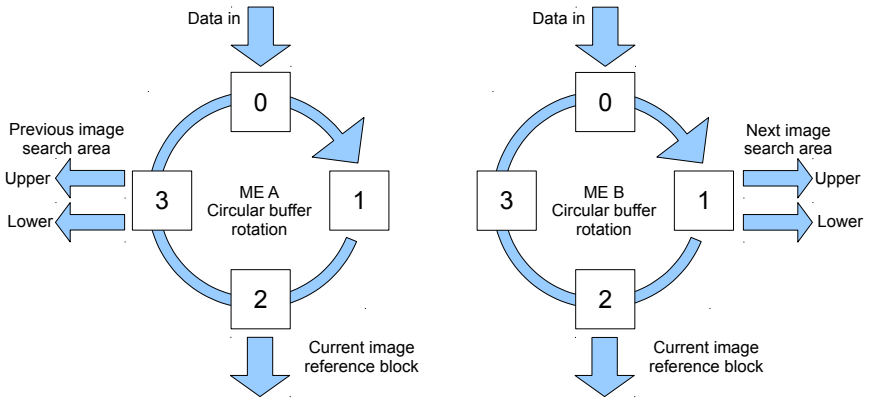


Figure 6.5.: ME A and B circular buffers, in the left for backwards ME, in the right for forwards ME

Three LMC_C2S (Section 3.4.5 on page 35) were used to fetch the streams with the order described above, however the order of the two most inner loops (column-wise, top to bottom, left to right) could not be met because of the burst oriented fashion (row-wise left to right in bursts of 16 pixels) that the CMC uses to deliver the data. Therefore, three LMC_APT (Section 3.4.4) were used to transpose the image blocks, which basically inverts the order of the two most inner loops, allowing the correct streams to be produced. The cascade of these two LMCs can be seen in Figure 6.6.

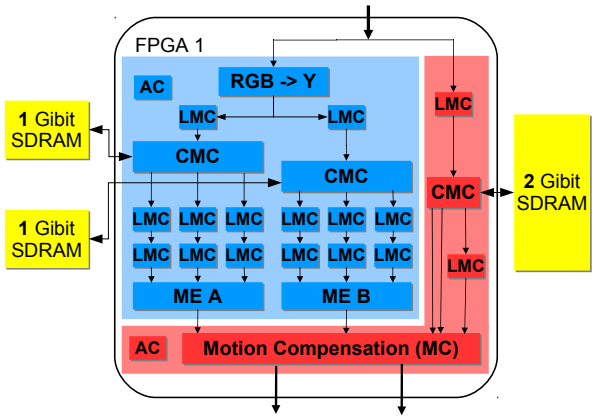


Figure 6.6.: ME and MC implementation using the FlexWAFE architecture

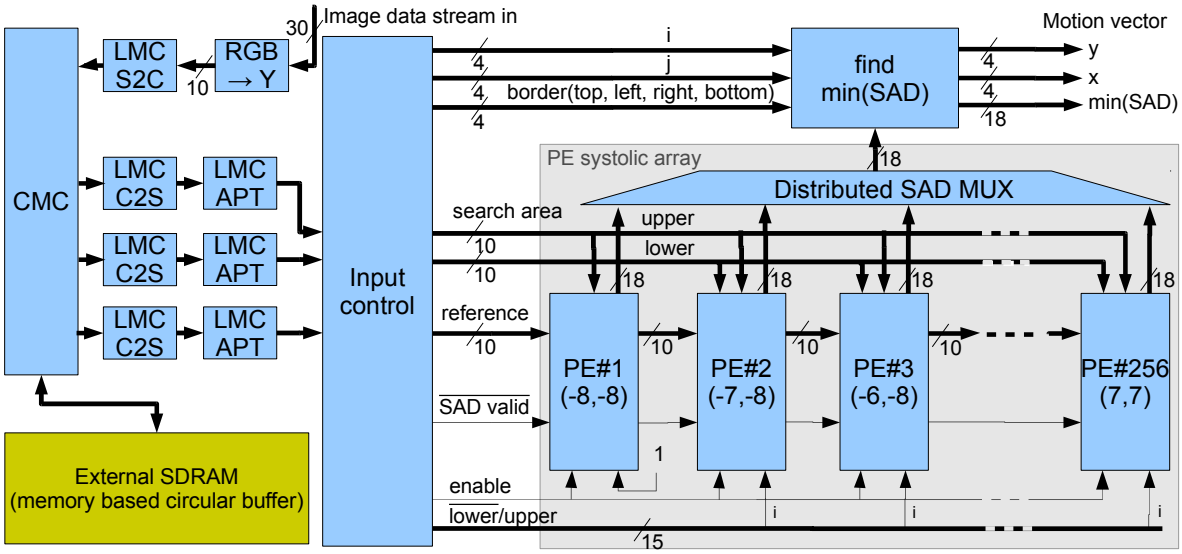


Figure 6.7.: Motion estimation functional block diagram for $r = p = 8$ and 10 bpp

The next task was to synchronize the three resulting streams. Each image block of *reference* and *search upper* contains $M \times N$ pixels, but each image block of *search lower* contains only $(M - 1) \times N$ pixels. Therefore, a LMC_sync_join could not be used here. Besides that, *reference* and *search upper* run synchronously to each other, but *search lower* runs with a delay of M relatively to the other two. For those reasons a custom block named *input controller* was developed that synchronizes the three streams output from the LMC_APT blocks before delivering them to the systolic array of PE elements. This block also takes *border effects* into consideration. Image blocks located at the top, left, right and bottom borders of the image do not allow all (i,j) combinations of MV because some of them would require search area pixels from outside of the image, and such pixels do not exist. Therefore, this block provides only pixels from image regions that can generate valid MVs. Figure 6.7 presents more details of the inter-block connections.

The Processing element implementation of each one of the 256 PEs of Figure 6.7 can be seen in Figure 6.8. The search area multiplexer, pixel subtraction, and reference pixel delay operations were hand-placed in the FPGA fabric so that two bits use a single slice. In our implementation the input pixels are 10 bit wide, so these operations use 5 slices. The absolute value calculation and the result accumulator operations were hand placed in the FPGA fabric so that again, two bits use a single slice. We need to accumulate 256 ($M \times N$) values of 10 bits, to do so 18 bits are required, which took 9 slices.

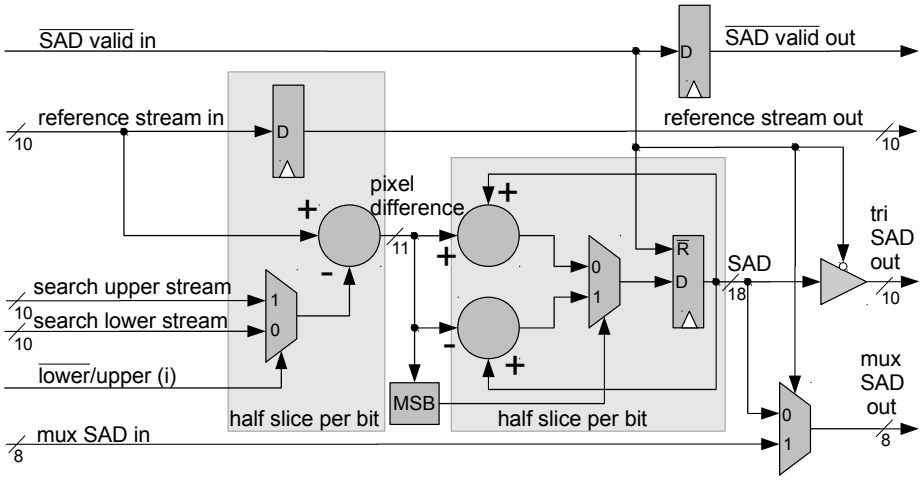


Figure 6.8.: SAD processing element (basic block of the ME systolic array) for $r = p = 8$ and 10 bpp

To find the minimum SAD a 256x18bits multiplexer is required. We distributed this multiplexer and spliced its implementation so that 10 bits are implemented using the 10 available tristate buffers in each PE (one tristate buffer per 2 slices). This forms a tristate bus where only one of the PEs is active at a time, effectively creating a 256x10bit multiplexer. This bus is not shown in Figure 6.8. The other 8 bits are multiplexed in a cascaded fashion through the PEs using 2x8bits multiplexers requiring 4 slices in each PE as shown in the bottom of Figure 6.8 (SAD in / SAD out). The logic described so far requires 18 slices and there is some extra control logic that occupies 2 slices, and that adds up to 20 slices or 5 configurable logic blocks (CLB) per PE. Their physical placement can be seen in the right side of Figure 6.9.

The simple cascaded 256x8bit multiplexer is very slow and does not meet the required timing. To make it faster we segmented it in eight groups of eight PEs each and the result of the each of the eight groups is fed into a tristate bus. Again, only one of the groups is exclusively active and that creates a 8x8bit multiplexer. This allows to meet timing at the expense of 256 extra tristate bus drivers, which is not an issue because each FPGA CLB contains two of those. Each group of 8 PEs would require 8x5 CLBs for their implementation plus 5 CLBs for the 8bit tristate mux, but that would leave most of the logic resources in the last 5 CLBs unused. To improve that, we grouped two groups of 8 PEs together and let them share the tristate buffers via a LUT based mux2 physically placed between the two groups. In the 16×16 image block implementation this saves 40 CLBs per ME array, which is a 5,5% area saving.

Resource	Usage	Percentage
RAMB	62 out of 232	26%
Slices	19,157 out of 23,616	81%
TBUF	5,408 out of 11,808	45%
FF	24,752 out of 47,232	52%

- bidirectional ME with block size 16x16
- bidirectional MC
- searches -8/+7 vector interval
- 24 fps @ 2048x2048, 10bpp (125 MHz)
- 1024 net add/sub operations/pixel
- 514 net comparations operations/pixel
- 155 net Goperations/s

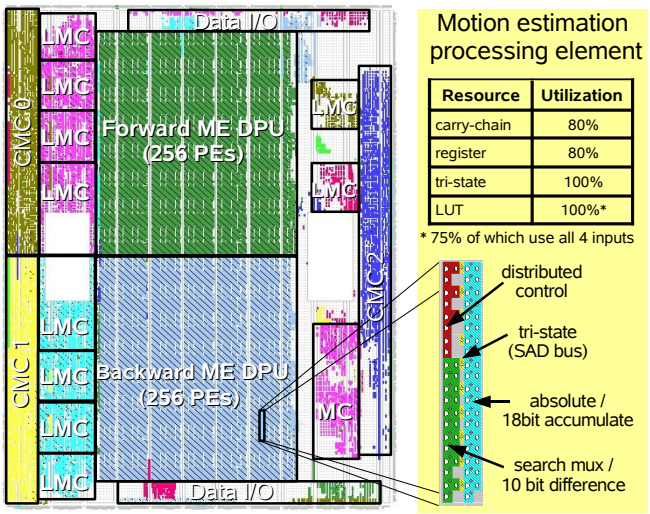


Figure 6.9.: ME and MC floor-plan on a FlexWAFE FPGA

The physical mapping of both backwards (ME A) and forwards (ME B) cores can be seen in the bottom and in the top of Figure 6.9 respectively. Their respective CMCs and LMCs are on the left side of the FPGA. The motion compensation is on the right side of the FPGA together with its CMC and its two LMCs.

6.2.2. Motion Compensation

Once the block displacements are found by the two ME engines, a new image is created whose content is similar to the reference image (buffer 2 in Figure 6.10), but it is assembled entirely of displaced blocks from either the previous (buffer 3 in Figure 6.10) or the following (buffer 1 in Figure 6.10) image. This step is known as bidirectional Motion Compensation (MC) because it compensates movement relative to the temporal previous (A) and to the following (B) image and works in the full RGB color space.

The difference between this newly created image and the current reference image is the film grain noise together with some small artifacts caused by imperfections of the motion estimation and compensation principle [42]. When the SAD value exceeds a *min_valid_SAD* threshold – which means that the two blocks differ too much and motion compensation should **not** be performed – the respective image block is replaced by the image block of the reference image. The advantage is that this avoids artifacts in the noise detection, the disadvantage is that no temporal information (consecutive image frames are stored in different spatial areas in the film, so this is in reality *film spatial* information) is used in the noise reduction of those blocks because their content is the same in the reference image. This threshold is typically set to a high value in order to avoid using pixels from the current reference image because doing so would reduce the effectively of the de-noising algorithm as a hole [42]. To further avoid artifacts created by imperfect motion estimation and compensation, each motion com-

pensated pixel is compared with its reference pixel (current image) and if it differs too much (> *max_pixel_diff*), the reference pixel is taken instead.

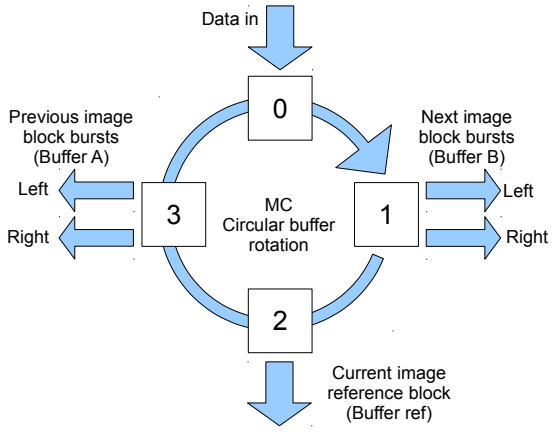


Figure 6.10.: MC circular buffer sequence

signal	**bit-width**	**description**
bpp	4	bits per pixel (constant 10 in this implementation)
M	4	image block height (constant 16 in this implementation)
N	4	image block width (constant 16 in this implementation)
W	12	maximum image width (constant 4096 in this implementation)
x	$\log_2$(N), 4 in this implementation	MV horizontal block displacement
y	$\log_2$(M), 4 in this implementation	MV vertical block displacement
SAD	bpp + $\log_2$(M.N), 18 in this implementation	Sum of absolute differences from the best matching MV
min_valid_SAD	bpp + $\log_2$(M.N), 18 in this implementation	SAD threshold, only SAD values below it are considered valid
SAD_OK	1	SAD valid flag, only blocks where this flag is valid are motion compensated
$\overline{A}/B$	1	0 use pixels from previous image, 1 use pixels from next image
max_pixel_diff	bpp, 10 in this implementation	*if* $\lvert MC_{pixel} - ref_{pixel}\rvert > max\ pixel\ diff$ use ref_{pixel} *else* use MC_{pixel}
image height	12	image frame height, must be a multiple of M
image width	12	image frame width, must be a multiple of N

Table 6.1.: Motion compensation constant and signal descriptions

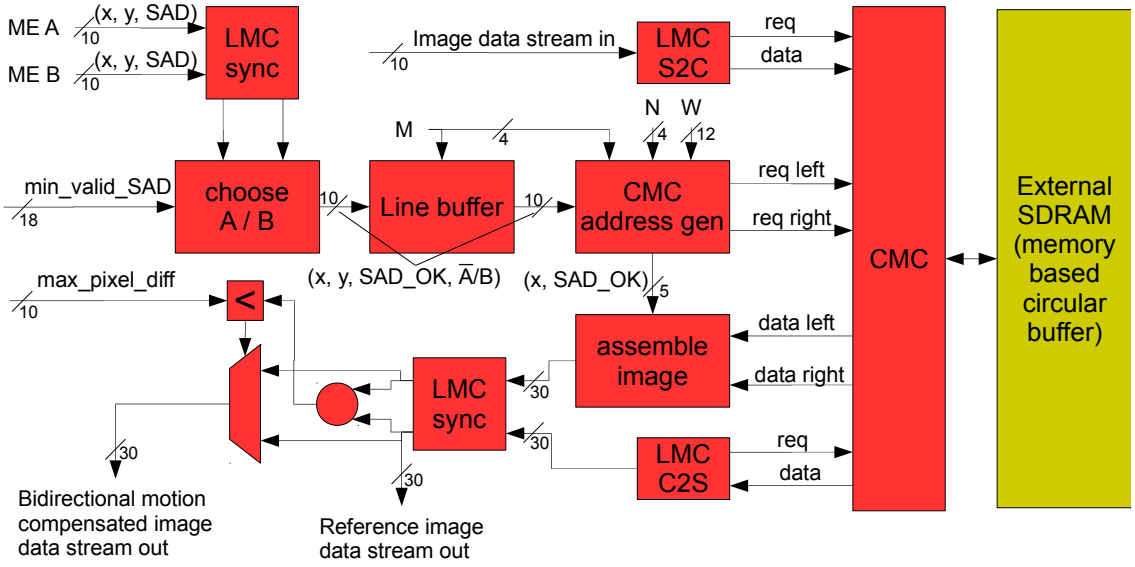


Figure 6.11.: Motion compensation functional block diagram

Figure 6.11 presents the functional block diagram of the motion compensation implementation. Table 6.1 describes the signals used in that figure. Each block will be explained in detail in the following paragraphs.

LMC sync join The output of each of the ME engines is a stream of vectors (x, y, SAD) as seen in the top left side of Figure 6.11. Motion compensation first synchronizes the two incoming streams of motion vector information. This is required because the address access patterns to external memory of the two ME engines are different (see Figure 6.5 on page 74) and the CMC response time depends on the patterns thereby causing small skews in the timing of the output motion vectors. This unit was originally developed for the DWT processing (Section 6.2.3) and its re-usage here proves that flexible reusable code saves development time. This LMC contains n stream inputs that can have different bit-widths and produces n streams outputs synchronized to each other by using n FIFO-similar internal memories. The LMC sync join instance used here uses $n = 2$.

Choose A/B After synchronization the SADs's of the two motion vectors streams are compared and a new stream is produced containing the motion vector with the smallest SAD, a SAD_OK flag indicating that the chosen MV has a SAD smaller than the *min_valid_SAD* threshold and a flag containing information about which of the two (forward/backward) motion vectors was chosen. For example for block-size of 16x16 pixels (M=16 and N=16), the stream consists of 4 bits for horizontal block displacement in pixels, 4 bits for vertical block displacement in pixels, 1 bit for the SAD_OK flag and one bit for the $\overline{A}/B$ (forward/backward) image flag. Hence the vector $(x, y, SAD\_OK, \overline{A}/B)$ has 10 bits.

Line buffer This stream is then stored in a slightly modified version of a LMC_APT. Like the LMC_APT this one has a internal memory divided into four regions that operate like a circular buffer (see Appendix B), and the input stream is written in ascending address order into one of these regions. Each motion vector line is written into a different region. The difference is that each region is read out not one but M times before switching to the next region. It has an extra programmable *repeat* register for this purpose. Therefore, the output stream of this LMC is a M periodic repetition of each input vector line. Our implementation can operate with image widths of up to 4096 pixels (limited only by the amount of external SDRAM for the circular buffers of Figure 6.10). The block width is 16 so each vector line will consist of up to 4096/16=256 vectors. To store those in four regions 256×4×10=10Kibits are required, which corresponds to a single FPGA 18Kibits BRAM.

CMC address gen The next step is to request the motion compensated image pixels from external memory. These pixels will come from the previous (left side of Figure 6.10) or from the next image (right side of Figure 6.10). If SAD_OK is false, no requests are performed because the pixels that would be requested would be discarded later anyways, so we just do not request them, which saves data-rate.

The vector information is image-block based but the assembly of the motion compensated image is image-line based. Furthermore, the image information is contained in external memory. To access it, requests must be made to a CMC that is burst oriented and delivers 16 consecutive pixels of an image line. The CMC does not allow unaligned accesses [91, 49]. This simplifies CMC design but complicates the assembly of the motion compensated images. When the x motion vector is 0 there is no horizontal displacement between the original image block and the motion compensated image block in the next or previous image. Then only one request is initiated using the *req left* port of Figure 6.11. All the three images are stored in the same way, which means that just like when accessing the original image it can be done in a pixel by pixel, burst by burst, image line by image line, the same can be done for the motion compensated pixels. This is valid when $burstlength \leq N$. The first implementation of this algorithm is described in [122]. Whenever the x motion vector is different from zero, two burst requests must be made to fetch 32 consecutive pixels (*req left* port fetches the first 16, *req right* port fetches the last 16) of which only 16 will be used and the rest must be discarded in the *assemble image* block. Which 16 pixels are to be kept is determined by the x value of the image block. The image is divided in blocks ($M \times N$) and has a maximum width of W pixels, A is buffer 3 in Figure 6.10 and B is buffer 1 in Figure 6.10. The algorithm written in pseudo-code is:

```
for m in 0 to imageheight/M
 for n in 0 to imagewidth/N
  for i in 0 to M
   if SAD_OK(m, n) == 1
    if A/B(m, n) == 0
     if x(m, n) == 0
      request left  = A[(m*M + y(m, n) + i)*W + (n*N)]
     else if x(m, n) < 0
```

```
      request left  = A[(m*M + y(m, n) + i)*W + ((n-1)*N)]
      request right = A[(m*M + y(m, n) + i)*W + (n*N)]
     else
      request left  = A[(m*M + y(m, n) + i)*W + (n*N)]
      request right = A[(m*M + y(m, n) + i)*W + ((n+1)*N)]
     end
    else
     if x(m, n) == 0
      request left  = B[(m*M + y(m, n) + i)*W + (n*N)]
     else if x(m, n) < 0
      request left  = B[(m*M + y(m, n) + i)*W + ((n-1)*N)]
      request right = B[(m*M + y(m, n) + i)*W + (n*N)]
     else
      request left  = B[(m*M + y(m, n) + i)*W + (n*N)]
      request right = B[(m*M + y(m, n) + i)*W + ((n+1)*N)]
     end
    end
   else
     make no requests
   end
  end for
 end for
end for
```

Because reading the external memory using a CMC can have a latency of up to 200 clock cycles [49], the read requests are done asynchronously from the motion compensated image assembly.

Assemble image The assembly step requires only the (x, SAD_OK) information (5 bit) from the *CMC address gen* block, it does not need to know from which (previous/next) image the pixels come, nor the y vertical displacement, nor does it need to know the SAD of the MV. So the transferred information between these blocks is minimized. Motion compensation image assembly only needs to selectively discard the pixels from the non burst-aligned image blocks (blocks which had $x \neq 0$). And when SAD_OK is false it outputs empty pixel information.

LMC C2S The next step of the noise reduction algorithm (Section 6.2.3) requires a reference image stream, which is the non-motion compensated version of the current image. A *LMC_C2S* (Section 3.4.5) is used for that purpose and is synced with the *LMC_S2C* and the *CMC address gen* via an AC (not depicted in Figure 6.11) to form the circular buffer (Appendix B) depicted in Figure 6.10

Final LMC sync join The final step is to output two image streams synchronized with each other: the compensated pixel stream and the reference or current pixel stream. This

step uses the SAD_OK information to block-wise replace the empty pixels in the motion compensated stream with ones from the reference stream by reusing an LMC_sync_join block. The *max_pixel_diff* parameter is used to pixel-wise replace miss-estimated pixels with pixels from the reference image.

The final motion estimation and compensation implementation mapping is shown in Figure 6.6 on page 75 and its physical FPGA floor-plan is shown in the right side of Figure 6.9 on page 77.

6.2.3. Discrete Wavelet Transform Based Filtering

Wavelets perform a good spacial-temporal separation of the signal components and this is used here to better separate the noise from the information. The discrete wavelet transformation used in the noise reduction algorithm has the following requirements:

- lossless - there can be no data loss in the transformation process, that means that no data rounding or truncation can occur.
- reversible - the transformation must be reversible so that chaining the direct with the inverse transformation must produce the same data at the output as the data at the input (when the noise reduction shrinkage function is disabled).
- linear - it should transform into a linear space, so that the noise reduction can operate in a smother way.
- simple hardware implementation - the transformation should not use too many hardware resources, it should fit in the available FPGAs (two XC2VP50 FPGAs in the case of the FlexFilm board implementation, the third FPGA is used for ME/MC and the fourth for I/O)
- real-time operation - it should process 2048x1568 pixel images at 24 frames per second or more, which means 77 Mpix/s or more.

Current FPGAs do not implement floating point operations efficiently and floating point arithmetic operations potentially suffer from truncation and rounding effects. So in order to meet the first requirement a wavelet transformation was chosen that does not use floating point operations. From the available wavelets [102] the LeGall 5/3 wavelet was chosen. Its coefficients can be easily adapted to map integers in integers by multiplying its original coefficient set so that all coefficients are themselves integer [129]. So let X_i be an uni-dimensional integer signal defined for a infinite set of points in time ($X_i \in \mathbb{N} \wedge i \in [-\infty, \infty]$). The wavelet transformation will decompose this signal into two signals a low-pass (L) and a high-pass (H). The low-pass resulting signal is only defined for even points, it is zero on odd points and the high-pass is only defined for odd points and it is zero at even points. Both are said to be decimated by two, because each will effectively contain half the number of samples of the input signal (assuming that the zeros are ignored). The decomposition parametric filter equations, also known as direct transformation parameters, are shown in Equation 6.7 and 6.8 on the facing page. They will transform the input signal X_i from the time domain into the wavelet space domain. This domain is also known as space-time domain because wavelet transformation preserves some of the characteristics of the original signal. These equations already take in

consideration the *decimation by two* that occurs after the filtering step. To up-sample by the integer factor N, simply insert $N-1$ zeros between x_i and x_{i+1} for all i. Down-sampling by M (also called decimation by M) is defined for $x \in \mathbf{C}^N \wedge M \in \mathbb{N}$ as taking every Mth sample, starting with sample zero.

$$L_{2n} = -\frac{1}{8}X_{2n-2} + \frac{1}{4}X_{2n-1} + \frac{3}{4}X_{2n} + \frac{1}{4}X_{2n+1} - \frac{1}{8}X_{2n+2}, \tag{6.7}$$

$$H_{2n+1} = -\frac{1}{2}X_{2n} + 1X_{2n+1} - \frac{1}{2}X_{2n+2} \tag{6.8}$$

The resulting signal in the wavelet domain consists of interleaving both L and H signals resulting in:

$$Y_{2n} = L_{2n}, Y_{2n+1} = H_{2n+1} \tag{6.9}$$

To revert the resulting signal from the wavelet domain back to the time domain, the inverse wavelet transformation is used, its equations (also known as analysis equations) are shown next. They operate on the L^* and H^* signals that are *up-sampled by two* versions of L and H respectively:

$$L_n^{-1} = \frac{1}{2}L^*_{n-1} + 1L^*_{n+0} - \frac{1}{2}L^*_{n+1}, \tag{6.10}$$

$$H_n^{-1} = -\frac{1}{8}H^*_{n-2} - \frac{1}{4}H^*_{n-1} + \frac{3}{4}H^*_{n+0} - \frac{1}{4}H^*_{n+1} - \frac{1}{8}H^*_{n+2} \tag{6.11}$$

And to obtain the final result both equations are added together in a final step:

$$X^*_n = L_n^{-1} + H_n^{-1} \tag{6.12}$$

The direct low pass and the inverse high pass equations have five coefficients. The direct high pass and the inverse low pass equations have three coefficients. Typical implementations of this transform use FIR filters whose coefficient's are the ones described in the equations above to convolve the input signals in a stream-oriented fashion. So let $f(z)$, $g(z)$, $f^{-1}(z)$ and $g^{-1}(z)$ be the functions described by equations 6.7 to 6.12, their block diagram is then depicted in Figure 6.12 on the next page.

The final addition operation (Equation 6.12) that is depicted in the right of Figure 6.12 on the next page can be optimized by combining equations 6.10, 6.11 and 6.12 together:

$$X^*_{2n} = -\frac{1}{4}H_{2n} + 1L_{2n+1} - \frac{1}{4}H_{2n+2}, \tag{6.13}$$

$$X^*_{2n+1} = -\frac{1}{8}L_{2n-2} + \frac{1}{2}H_{2n-1} + \frac{3}{4}L_{2n} + \frac{1}{2}H_{2n+1} - \frac{1}{8}L_{2n+2} \tag{6.14}$$

The first, single decomposition-level implementation of this algorithm was done by [130] and later optimized and extended as described in the following paragraphs.

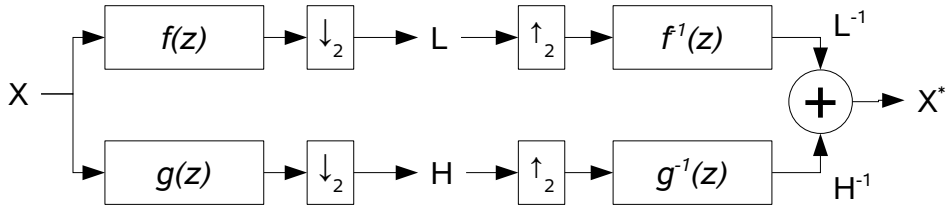


Figure 6.12.: Convolution based direct DWT followed by inverse DWT

Avoiding floating point operations As explained before the equations above can be multiplied by constants such that all filter coefficients become integer. By multiplying L by eight and H by two we get:

$$8L_{2n} = -1X_{2n-2} + 2X_{2n-1} + 6X_{2n} + 2X_{2n+1} - 1X_{2n+2}, \quad (6.15)$$

$$2H_{2n+1} = -1X_{2n} + 2X_{2n+1} - 1X_{2n+2} \quad (6.16)$$

And for the inverse transformation

$$64X^*_{2n} = -8(2H_{2n}) + 8(8L_{2n+1}) - 8(2H_{2n+2}), \quad (6.17)$$

$$64X^*_{2n+1} = -8L_{2n-2} + 16(2H_{2n-1}) + 6(8L_{2n}) + 16(2H_{2n+1}) - 8L_{2n+2} \quad (6.18)$$

This guaranties that all operands and coefficients are integer, and to recover the original signal the X^* signal must simply be divided by 64. That operation can trivially be done be truncating the six Least Significant Bits (LSB).

Multi-dimensional The wavelet-transformation operates on two-dimensional images by transforming the pixel lines (the horizontal direction) as an uni-dimensional signal and by transforming the resulting pixel columns (the vertical direction, again an uni-dimensional signal) to form the two-dimensional image transform. This works because the 2D DWT is separable. The image pixels are transferred between DPUs in streams like explained in chapter 3, pixel-by-pixel (from left to right), line-by-line (row-wise from top to bottom). The horizontal DWT is performed by a DPU that receives the 2D image stream and produces the output with the same order; both input, processing and output is done row-wise.

The vertical DWT is done by a different DPU although the operations performed are the same as the horizontal DPUs. But these operations are to be performed column-wise to an input stream that is row-wise and must produce a row-wise output stream. One solution would be to transpose the entire image, re-use the horizontal DPU and transpose the resulting image. That solution was not chosen because it would take big amounts of memory (two entire images of MxN pixels) and processing time. We transpose only the part of the image required by the transform (5 pixels in the case of the 5/3 wavelet), this only requires four cascaded line delays (4xM pixels). The line delays are done using four *line delay* LMCs, the transform is then

applied to their output and to the input pixel (five pixels total) and the output is produced row-wise. This minimizes memory requirements and reduces latency. The vertical DPU contains the line delays and the processing logic to calculate the column-wise DWT.

Wavelet based filtering is not only performed in the space-domain (horizontally and vertically like explained above) but also in the temporal domain. In the temporal domain a simpler, two-coefficients, Haar wavelet was chosen because only two consecutive images are to be used. One of these images is the current image, and the other image is the result of the motion compensation algorithm from Section 6.2.2 on page 77. The two output images of this Haar wavelet are then 2D filtered using 5/3 wavelets, the two resulting images are transformed back using an inverse Haar wavelet as depicted in Figure 6.2 on page 70 and the outcome is the final noise-reduced image.

Multi-level To improve the results of the noise-reduction algorithm, three consecutive 2D direct 5/3 wavelet transforms were performed, followed by the corresponding three inverse transforms. Each transformation is referred to as a *transformation level*, the first one is level one, the second is level two and the third is level three. Every 2D DDWT level decomposes one MxN pixel image into four sub-band images: LL, LH, HL, HH of $M/2$x$N/2$ pixels each. Only the LL (horizontally and vertically low-pass) sub-band image is used as input to the next level, the other three are filtered using shrinkage [42] filters. The next 2D DWT level decomposes the $M/2$x$N/2$ LL image into four $M/4$x$N/4$ images and so on. The 2D IDWT process is similar, each level inputs four sub-band images of $M/2$x$N/2$ pixels each and produces one image of MxN pixels. Figure 6.13 depicts this three level structure using FIR filters to implement the 5/3 wavelet transformations.

Multi-channel Color images are typically represented digitally using a RGB triplet or a RGB +Alpha quadruple. So we had to build three filtering channels, one for each color component. We simplified the design by developing a single channel and instantiating it three times in hardware. This was done with the FIR based DWT filters, later on the project we used a different approach that delivered better results. This other approach will be detailed in the following paragraphs.

Dynamic range Equations 6.15 to 6.18 are easily implementable in FPGA hardware and provide full precision by avoiding truncation on all stages except in the very last division by 64. But they do it by increasing the number of bits required to represent their intermediate signals. This issue becomes of greater importance once we consider the multi-level DWT approach in the noise reduction algorithm. Figure 6.13 on the following page shows the cascading of horizontal and vertical FIR based filters following the same scheme depicted in Figure 6.12 on the preceding page and replicated three times to achieve a three level DWT decomposition. Each signal is annotated with the bitwidth it requires, assuming an input bitwidth of 10 bits per pixel. Section D.2 explains the theory and the calculations used to determine the intermediate bitwidths.

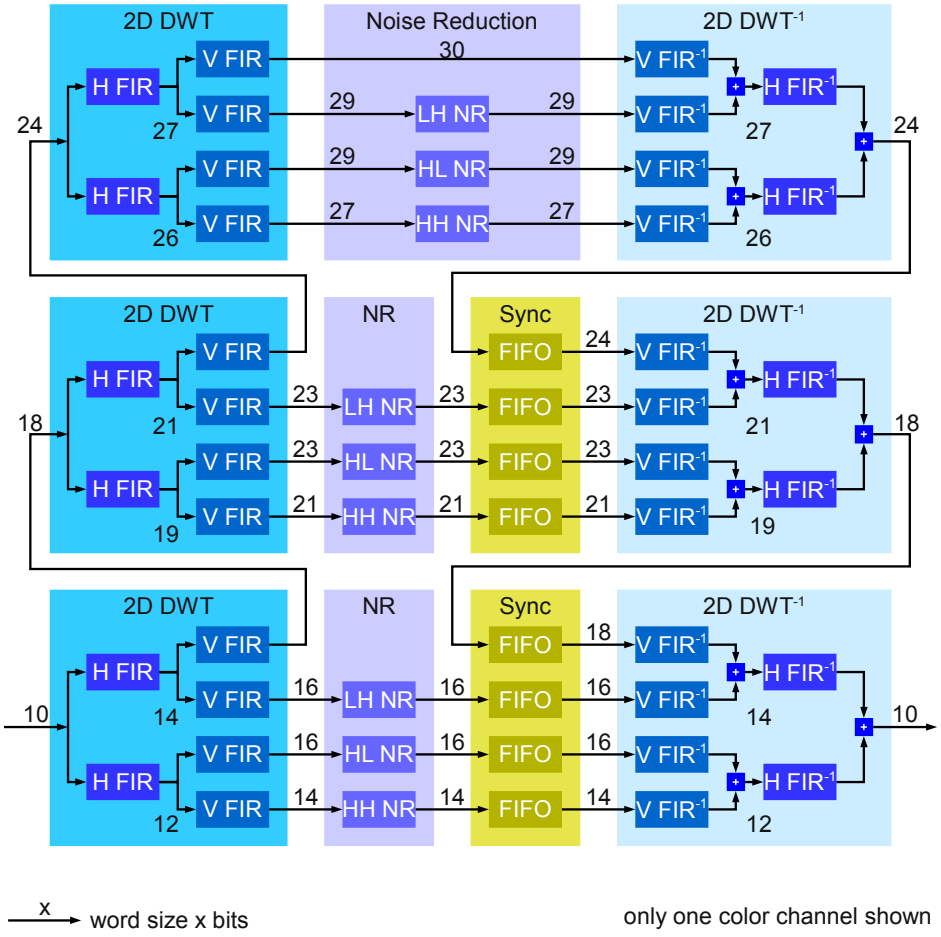


Figure 6.13.: FIR based 2D DWT/IDWT noise reduction scheme

Lifting based DWT implementation The forward DWT equations (6.7 and 6.8) can be respectively rewritten in the form of:

$$H_{2n+1} = -\frac{1}{2}X_{2n} + X_{2n+1} - \frac{1}{2}X_{2n+2}, \tag{6.19}$$

$$L_{2n} = +\frac{1}{4}H_{2n-1} + X_{2n} + \frac{1}{4}H_{2n+1} \tag{6.20}$$

The second equation depends not only on the input signal but also on the result of the first equation.

As for the inverse equations (6.13 and 6.14) their lifting counterpart is respectively:

$$X^*_{2n} = -\frac{1}{4}H_{2n-1} + L_{2n} - \frac{1}{4}H_{2n+1}, \tag{6.21}$$

$$X^*_{2n+1} = +\frac{1}{2}L_{2n} + H_{2n+1} + \frac{1}{2}L_{2n+2} \tag{6.22}$$

Equation 6.19 and 6.21 are referred to as predict equations, and Equation 6.20 and 6.22 as update equations, because they update/refine the result of the predict equations. This dependency is clearly depicted in Figure 6.15 on page 89.

As can be easily seen, the lifting based direct DWT and its inverse require 8 addition/subtraction operations and are therefore more efficient than the FIR based approach that requires 13 addition/subtraction operations.

One other advantage of the lifting based implementation, when compared with the FIR approach, is that the Symmetrical Periodic Extension (SPE) is easier to implement. Figure 6.14 on the next page displays the implementation of the horizontal 1D direct DWT (DDWT) and inverse DWT (IDWT). To save hardware resources when implementing the three level 2D transformations, a further optimization will be introduced in the next paragraph.

SPE for finite signals Images are finite two-dimensional signals, and correct reconstruction using DDWT followed by IDWT is only possible for infinite signals. So we need to transform our images into infinite size. To do so we could simply make them periodic by replicating them vertically and horizontally and them combining them into a single infinite 2D signal. This however, is not possible because it would require infinite memory and processing power to produce the resulting infinite image. But one property of the convolution comes in our help: convolution of infinite periodic signals (our replicated input image) with a finite signal (the wavelets) produces a infinite periodic signal. Because the resulting signal is periodic, we only need to calculate one period of it, all others will be identical. And to calculate it, we only need one period of the 2D input signal extended by some samples to the left, right, top and bottom, in order for the convolution to operate. The amount of samples to extend in each direction is dependent on the filter length, in this case on the wavelet used. So it is possible to use a finite input signal X, extend it by some samples producing signal X' and calculate its 2D DDWT Y by convolution operations with the desired wavelet. After that, Y can

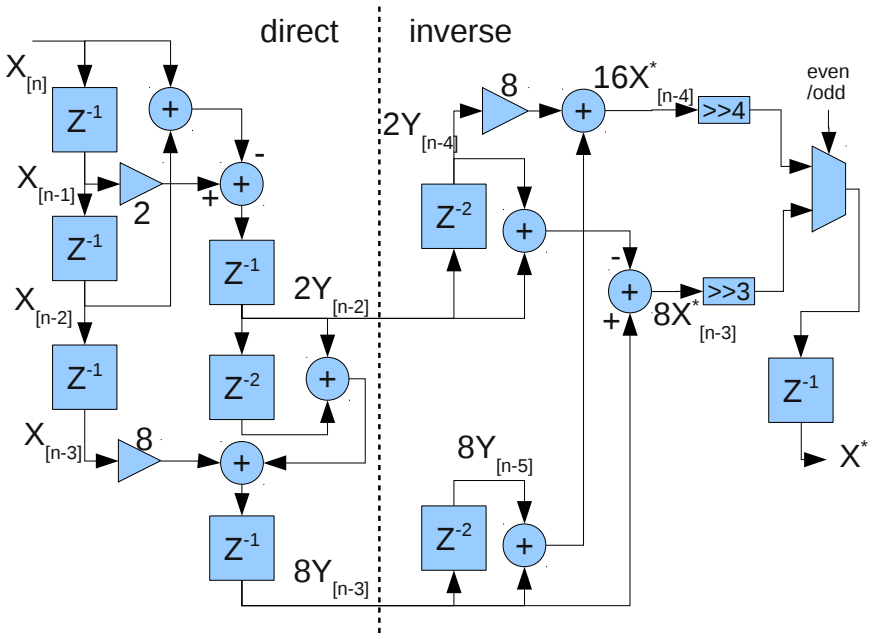


Figure 6.14.: Lifting based direct 1D DWT (left side) followed by inverse 1D DWT (right side) using all integer filter coefficients

be extended creating Y' and its IDWT X^* can be calculated by convolution with the inverse wavelet. In the end, $X == X^*$ although X is finite, this was possible because the auxiliary signals X' and Y' are a subset of the infinite periodic images that we only partially created. This is called periodic signal extension. The advantage of this technique is that we used finite memory and computational resources to calculate something that theoretically would require infinite resources. Figure 6.15 on the facing page presents an example of a one dimensional signal of length 6 (X) that has been extended one token to the right ($X_{[n+6]}$) to produce the wavelet space signal Y of length six using the coefficients depicted on Table 6.2. That signal was then extended one token to the left ($Y_{[n-1]}$) and to the right ($X^*_{[n+6]}$) to be able to compute the inverse DWT X^*. Note that $X^*_{[n+5]}$ depends on the $X^*_{[n+6]}$, this non-causal behavior (see causality) is a side effect of the lifting implementation.

One other simplification is that instead of extending the input signals X and Y, we can calculate the convolution on the edges of the images in a different way. All convolution operations that would operate on the extended samples are replaced by slightly different operations, all convolution operations that operate on the non extended pixels of X and Y are kept unchanged. By doing so, there is no need to create the X' and Y' images, it saves us the storage and computational resources required to create them. To change the convolution operations on the edges of the images only a small number of extra multiplexers is required for both FIR filter based implementations and lifting based implementations. Figure 6.16 on page 90 shows the four extra multiplexers (2a, 2c, 2f and 2g arcs multiplex between a/2a, c/2c, f/2f and g/2g respectively) required to transform the filter implementation in Figure 6.15 on the facing page into a self contained implementation without the need for data length extensions.

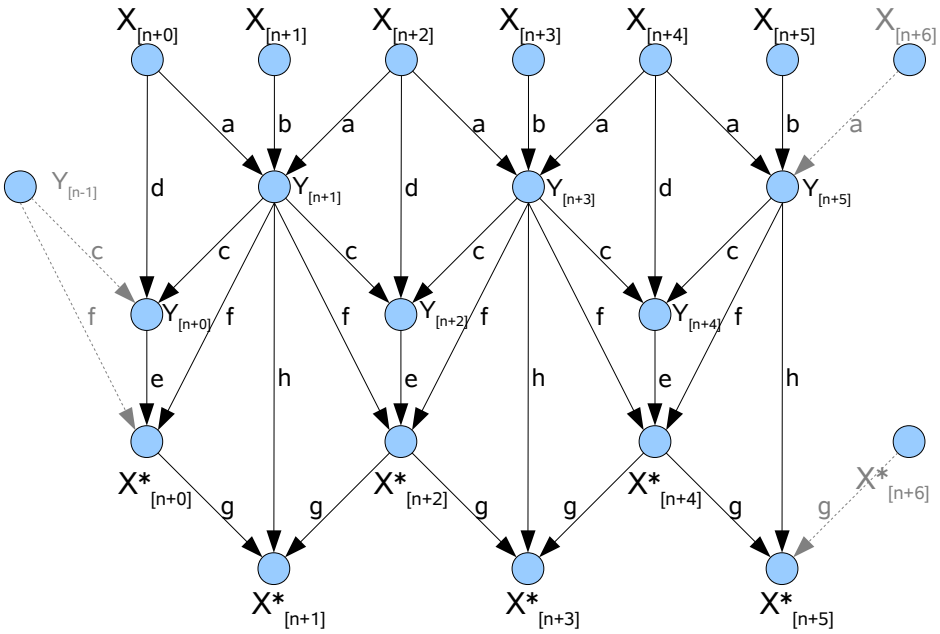


Figure 6.15.: one-dimensional lifting based direct DWT followed by inverse DWT

There are several algorithmic nuances to do this periodic signal extension [15]. We chose half-point Symmetrical Periodic Extension (SPE). It allows simple implementation especially when combined with the lifting scheme presented earlier.

Extension to integer-to-integer By adding 2 in certain operations, and by truncating in other operations it is possible to transform integers into integers although using fractional filter coefficients. The advantage is that the dynamic range of the intermediate results is greatly reduced, and the final precision is still kept. So lossless operation is achieved with less bits, and therefore even when accounting the extra *add two* additions, less hardware is required. The filter coefficients are summarized in Table 6.2 on the next page

$$Y_{2n+1} = X_{2n+1} - \lfloor (X_{2n} + X_{2n+2})/2 \rfloor, \tag{6.23}$$
$$Y_{2n} = X_{2n} + \lfloor (Y_{2n-1} + Y_{2n+1} + 2)/4 \rfloor \tag{6.24}$$

$$X^*_{2n} = Y_{2n} - \lfloor (Y_{2n-1} + Y_{2n+1} + 2)/4 \rfloor, \tag{6.25}$$
$$X^*_{2n+1} = Y_{2n+1} + \lfloor (X^*_{2n} + X^*_{2n+2})/2 \rfloor \tag{6.26}$$

This will then change the implementation in Figure 6.14 on the facing page into the optimized implementation depicted in Figure 6.17 on the next page.

This change greatly reduces the amount of bits necessary to store the intermediate results, therefore reducing the FPGA area and potentially increasing the maximum frequency at which

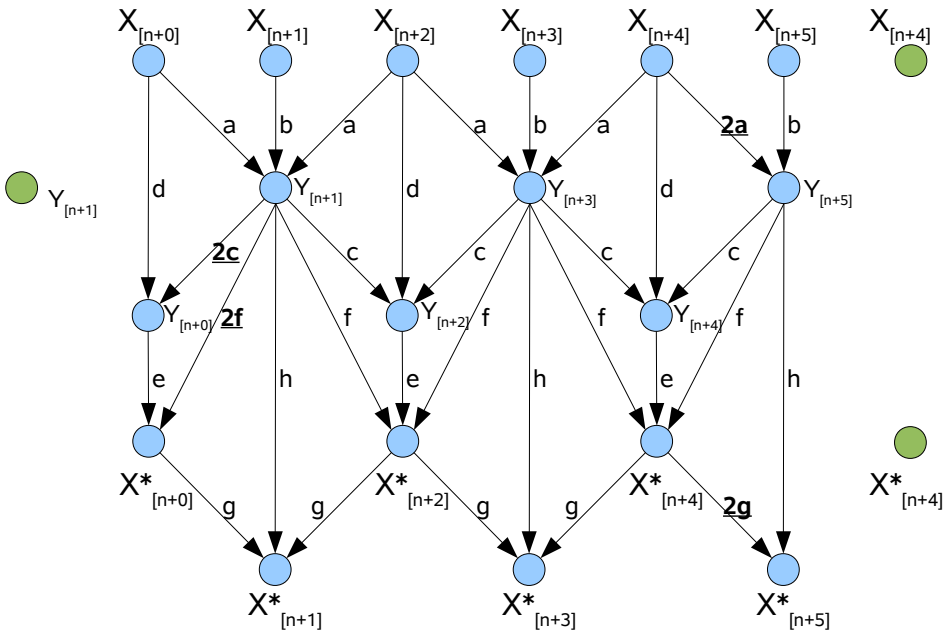


Figure 6.16.: one-dimensional lifting based DWT/DWT^{-1} with the four multiplexers required for SPE (highlighted in bold) and the mirrored data sample points (in green)

type \ parameter	a	b	c	d	e	f	g	h
Integer to integer	-1/2	1	1/4	1	1	-1/4	1/2	1
other	-1	2	1	4	4	-1	1	2

Table 6.2.: DWT filter coefficients

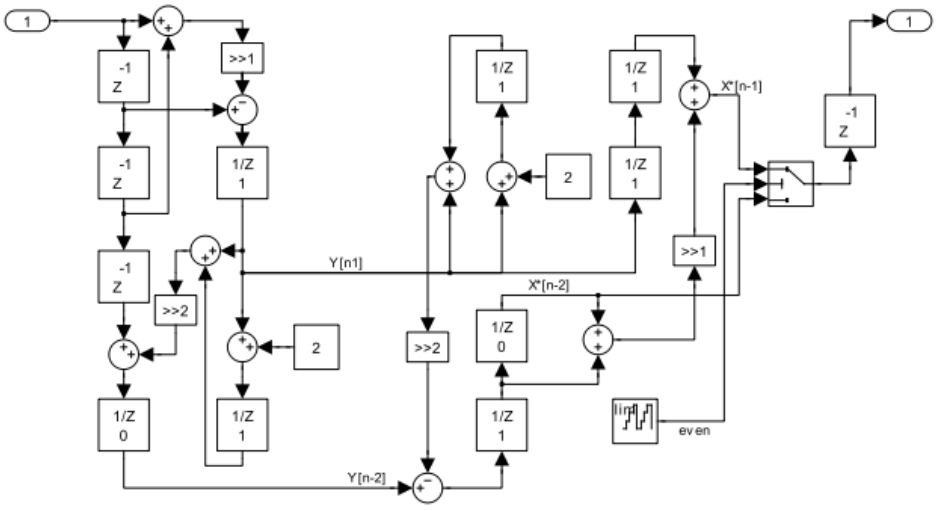


Figure 6.17.: Simulink block diagram of lifting based direct 1D DWT (left side) followed by inverse 1D DWT (right side) using fractional filter coefficients and truncation.

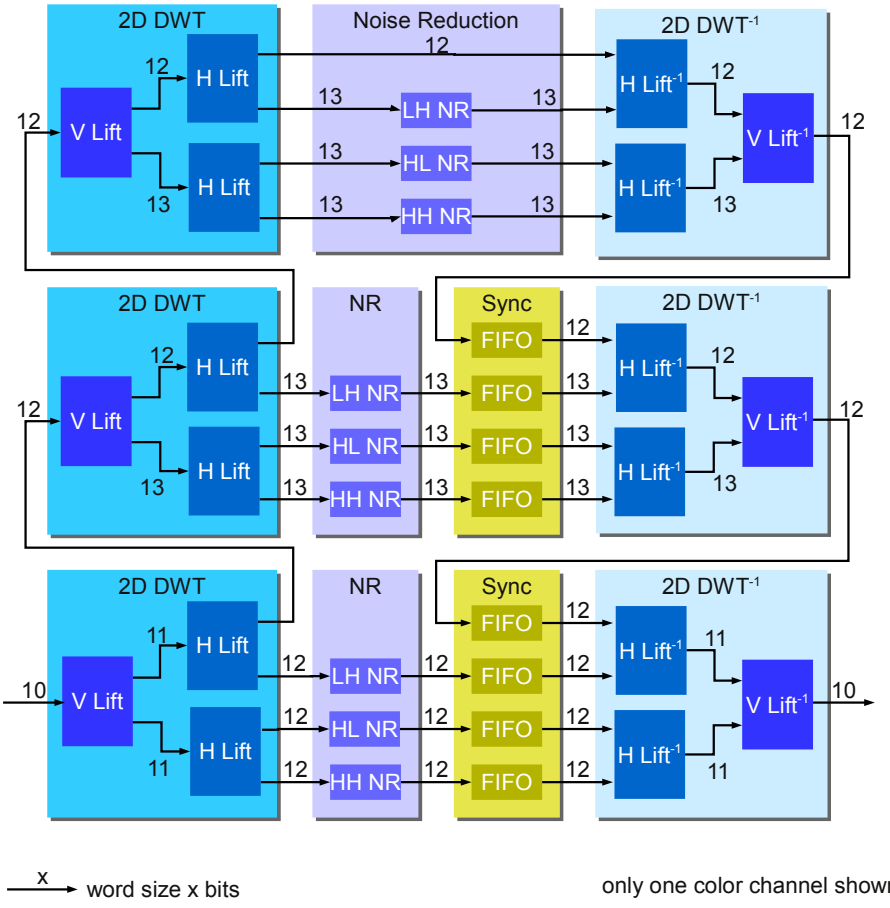


Figure 6.18.: Lifting based direct 2D three level DWT (left side), filtering and buffering in the wavelet space (center), followed by inverse 2D three level DWT (right side) using fractional filter coefficients, lossless truncation and taking filter cascading effects into consideration.

the filters can operate. This eases up the effort that the Xilinx ISE tool requires to place and route the design, allowing it to achieve timing closure faster.

The resulting widths were calculated by the technique presented in Section D.2 on page 116 and are presented in Figure 6.18

Latency The latency of the horizontal filters is close to the theoretical minimum. Assuming one new pixel is available per clock cycle, for a 3/5 DWT the minimum would be three clock cycles for the high-pass, and 5 clock cycles for the low pass. In the lifting implementation, it is three cycles for the HP, and four cycles for the LP, and that already takes in account the extra addition and truncation operations required for integer-to-integer operation. The LP latency is lower than the theoretical minimum because of the SPE, which allows to output data sooner.

The inverse transformation has three clock cycles latency. The vertical filters have the same latencies, but in image lines and not in pixels (samples).

On the second DWT level, the horizontal direct filters will have the same latency, but their input stream has been decimated by two, so their latency relative to their input stream is $2^{L-1}(K-1)+1$, where K is the filter latency (three for the predict step, four for the update step) and L is the DWT decomposition level, 1, 2 or 3 in our application. On the third DWT level the latencies of the direct filters will be $4(K-1)+1$ for the same reasons stated above. Again the vertical filters will have the same latencies in image lines.

Results The objective is to filter image sequences in the wavelet space, therefore filters are introduced between the direct and the inverse transformations. Because each stream will have different latencies depending on the path taken on the multilevel filters, FIFO buffers have to be introduced between direct and inverse transformations in order to synchronize the streams. In order to minimize these buffers, and the image line transposition memories in the vertical filters, it was decided to do the vertical filtering first followed by the horizontal. This has also the advantage that the horizontal filters use less resources than the vertical ones, and this approach uses less vertical filters than horizontal filters. The result can be seen in Figure 6.18.

For multilevel operation, multiple decomposition levels are cascaded, and their outputs are fed to an also cascaded set of inverse transformation filters as depicted in Figure 6.18. For the noise reduction application shrinkage blocks are inserted in between so that the noise reduction happens in the wavelet space. These NR blocks have a latency of one clock cycle, and can process one pixel per cycle. As described before the filters have latencies that depend on their nature (horizontal or vertical, direct or inverse). To achieve perfect reconstruction (lossless operation) the streams at the two inputs of each and every inverse filter must be synchronous to each other. Because each stream follows different paths with different latencies, it is necessary to synchronize them before feeding them to the inverse filters. That task is performed by the *sync FIFO* blocks in Figure 6.18. The *sync FIFO* at the center of the figure is responsible for synchronizing the level 2 filtered results (LH, HL, HH streams) with the LL stream that traveled through the direct and inverse filters of level3 and therefore has the latency of the combined filters of that level. Assuming an input image of width *W* pixels and image height of *H* pixels, the latency of the LL stream relative to the other 3 streams at level 2 is:

$$2\times 4 + 8\times \frac{W}{8}[pixels] \tag{6.27}$$

That amount is small enough to fit in *BRAM* memory blocks existing inside the XC2V50P FPGAs.

The *sync FIFO* in the bottom of Figure 6.18 are much bigger and can not be implemented exclusively with memory blocks inside the FPGA, external memory must be used.

The LH, HL and HH filtered streams produced at the first level have been decimated by two on the vertical and on the horizontal direction. Figure 6.19 shows the timing of these streams when an input image of 4x8 pixels is input. The horizontal decimation by two causes the small stream gaps of one pixel, the vertical decimation the bigger gaps of four pixels (matching the image width).

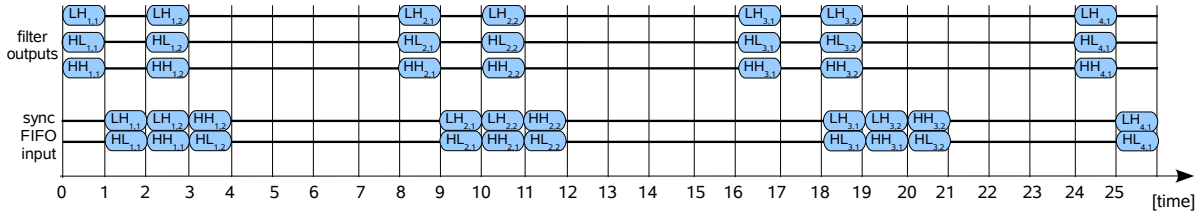


Figure 6.19.: three streams output from the filters in the top, and one stream (composed of two 16bit sub-streams) input to the synchronization FIFOs in the bottom

Each of the three streams has a precision of 16 or less bits (assuming a input precision of 10 bits), the external SDRAM interface has 32 or 64 bits width. To minimize the amount of the CMC ports, the decimated nature of the three streams is used, two streams are packed together into a 32 bit word. Their order can be seen in the bottom of Figure 6.19. This packaging scheme minimizes the amount of multiplexers and de-multiplexers required to implement it. It consists of three periodic steps: LH-HL, LH-HH, HH-HL. This effectively transforms three, decimated-by-two, streams of 16 or less pixels each, into one 32 bits non-decimated stream. This imposes that the original input image width must be a multiple of four, but that is guaranteed because the three levels of decomposition require a multiple of eight image width and height. So when the original image is $W \times H$ the packed stream has a width of $3 \times W/4$ [32bit-tokens] and a height of $H/2$ (due to the vertical decimation). The packed stream is non-decimated horizontally, but it is decimated by two vertically, hence the big gaps at the bottom of Figure 6.19.

6.2.4. Multi-level DWT based NR implementation

At a first filter stage the difference between the reference and motion compensated image is calculated using a Haar wavelet. The Haar wavelet produces two image streams: the low-pass image (the average of both images) and the high-pass image (the difference between the two images and therefore contains mainly film grain).

The following two DWT filters are applied in parallel to both image streams. Each filter level consists of three steps: DDWT, noise reduction, IDWT (Figure 6.13 on page 86 for FIR based, Figure 6.18 on page 91 for lifting based implementation). Each color channel is filtered independently which results in three separate filter channels for each image stream.

The first step is the three-level DDWT which performs a wavelet transformation of the entire image. Each level of the transformation consists of cascaded FIR filters in the horizontal and vertical image direction. Each filter outputs one high-pass and one low pass stream however, since each stream is decimated by two, the total number of output pixels equals the amount of input pixels. Three of these levels are cascaded to produce an improved spatial-temporal information separation by feeding the low pass output (sub-band) into the next level.

At this point the remaining three sub-bands of each level contain the image noise in the form of low pixel values (low energy), while the high pixel values (high energy) contain the real

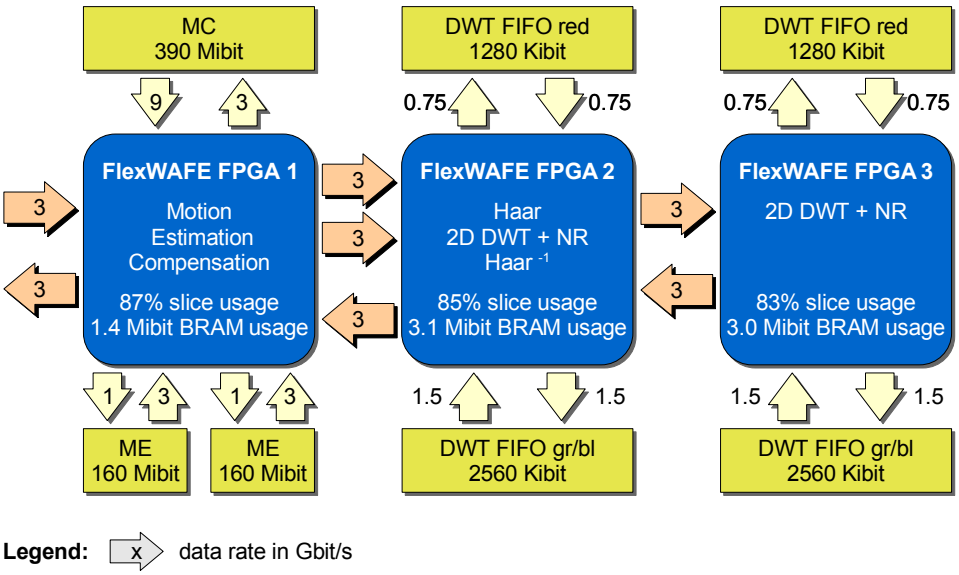


Figure 6.20.: Noise and grain reduction application, FPGA mapping

image information. The noise is reduced by applying a soft shrinkage function with a variable, user-definable threshold which replaces absolute pixel values below the threshold with zero.

After that, the image is reconstructed from the shrunk sub-bands by using a 2D three-level IDWT filter which consists of FIR filters similar in structure to the ones on the DDWT.

Both images from each DWT filter are then combined by an inverse Haar wavelet to form the final, filtered image.

6.2.5. Additional Resources

To be able to change parameters at run-time by the host without disturbing the image stream transport, an additional low data rate control bus from the IO-FPGA to the FlexWAFE-FPGAs (16 bit bidirectional data, 4 Kword address space per FlexWAFE FPGA) is provided.

Additional resources are a runtime-programmable parameter flash memory for the IO-FPGA, temperature/voltage monitoring units for each FPGA and Joint Test Action Group (JTAG) chains for programming the IO-FPGA configuration flash and the FPGAs if the board is operated stand-alone. These resources are not shown in the block diagram were discussed in Section 3.7.3 on page 44.

6.2.6. NR Application Mapping into FlexFilm Board

Figure 6.20 shows the mapping of the reference algorithm onto the FPGAs of the FlexFilm board. FlexWAFE FPGA 1 contains the RGB →Y conversion, ME and MC; FPGA 2 the Haar and inverse Haar filters and one DWT filter; and FlexWAFE FPGA 3 contains the second DWT.

Additionally, the required minimum data rates for real-time processing at 2K×2K resolution are given. As can be seen, from FlexWAFE FPGA 1 and 2 two streams need to be transported – the reference and the compensated image – while the remaining connections transport a single stream.

The required external SDRAM capacity and data rate requirements are also shown. Both ME and MC require large frame buffers because several frames need to be stored due to the complex address patterns. The DWT filters require smaller buffers to compensate for the additional processing latency of the higher filter levels.

The I/O FPGA is not shown in the figure and simply forwards the incoming stream from PCIe to the FlexWAFE FPGA one and vice-verse.

6.2.7. NR Application summary

In the FlexFilm implementation the device driver allows faster than real-time (2K @>24fps) image transfer to and from the FlexFilm board using DMA, accessing the Control Bus and thus partially or completely reprogramming all the ACs at 500KB/s and reprogramming each FPGA individually in 300 ms. The complex image processing algorithm developed by TU-Ilmenau [42] in the context of the FlexFilm project was implemented in hardware using the FlexWAFE architecture and work-flow. The algorithm is composed of three sub-algorithms that were partially implemented in parallel and that were explained in individual subsections of this chapter.

Motion estimation was fully parallelized using two engines of 256 similar processing elements placed physically and logically in an array form in the FPGA. Each array required only 17, single bit, external control signals, and was fed with 30 bit data-tokens. Each pipelined processing element runs at 125MHz, performed 4 operations per clock cycle, and the array can operate back-to-back without any idle cycles between pixels or frames. The maximal sustained performance of the data-paths contained in the arrays alone[4] was $2{\times}256{\times}4{\times}125M = 256$Gop/s. All operations were either 10 or 18 bit fixed point operations. The practical performance was only 65,4% of the peak (2K @ 26fps → 167,5Gop/s) because of pipeline stalls caused by waiting for SDRAM external data via the CMC. The processing elements are strongly connected with each other, and the maximal data transfers between them is 30Kbit per clock cycle (2 × 256 × Figure 6.8) or 3,7Tbit/s. The logic that feeds the arrays with data and computes the motion vectors using the information calculated by the arrays was reused from the FlexWAFE library (LMCs and CMCs). Only 10% of it (input_controller and find_min in Figure 6.7) had to be written from scratch for the motion estimation application.

Motion compensation uses the MV obtained by the previous step to assemble an image that is similar to the current one but it is built from displaced blocks from the previous and the next image. Using both previous and next images has the advantage that at scene cuts at least one of the images relates to the current image (assuming that one scene has at least

[4] excludes all the array-external control logic

two frames) and the motion can therefore be compensated. Film grain is different from frame to frame, but the objects contained in these two images should be identical. So by filtering these two images it is possible to extract most of the film grain noise from the current image. Around 30% of the blocks had to be hand-coded, and the rest was reused from FlexWAFE. The external SDRAM access patterns are image content dependent and can not be predicted. Nevertheless the CMC was able provide sufficient data-rate so that MC was faster than the ME. Both ME engines and the MC were fit in the same FPGA as depicted in Figure 6.9.

Discrete Wavelet Transform was used to convert the images to the wavelet space where it's easier to remove the film grain noise from the image. First a Haar wavelet was used between the current frame and the motion compensated one. The two resulting frames were then transformed using 5/3 three levels wavelets and each of the resulting sub-bands were filtered using independent soft-shrinkage thresholds. The result was then inverse transformed, first with 5/3 and after that with Haar inverse wavelets. This process had a lot of reuse, four filters were coded: horizontal and vertical DDWT and horiz. and vert. IDWT; that where then instantiated multiple times for the three DWT levels and color components.

The transformations were firstly implemented using FIR filter structures due to strict requirements. On a latter step one requirement was relaxed a bit, by introducing a small non-linearity (the +2 addition and subsequent truncation explained in the integer-to-integer paragraph at Section 6.2.3), smaller than the non-linearity caused by the soft-shrinkage NR step (typically the soft shrinkage adds and subtracts numbers much bigger than 2). This paved way to a lifting based implementation of the filters that reduced the required FPGA area to a quarter of the original FIR based implementation. The new implementation greatly reduced the dynamic range (bitwidth) of the intermediate results therefore reducing the logic required to do the data-path computations. So instead of two FPGAs dedicated to DWT filtering (Figure 6.20 on page 94 has a FIR based DWT implementation), just a little bit over half a FPGA (13272 slices) was required.

6.3. FlexWAFE use in the noise reduction application

A total of 35 LMCs were used in the the application described in the previous sections. Fourteen of which were used to create circular buffers in external SDRAM. They (LMC_C2S and LMC_S2C) were programmed with four parameter-sets, one for each of the buffers, each buffer was the size of a 2K image. Four of them also performed special operations at the top or at the bottom of the images and used eight parameter-sets instead of four. For real-time operation they needed to change parameter-sets 24 times per second. They were however designed with much higher parameter-set change rates in mind.

Six others LMC_APT were used to reorder data-streams. Two of them repeatedly used a single parameter-set for every block of 16x16 pixels. The other four use one set for the top/bottom 16x16 row and another for the rest of the image.

Six LMC_external_FIFO were used to create SDRAM based FIFOs. Nine LMC_sync_join were used to synchronize streams. These are not run-time configurable and are not controlled by an AC.

Each DWT FPGA used 108 weakly-programmable filters. 216 are FIR or lifting filters, and 27 are soft-shrinkage filters.

Six ACs where used to control the LMCs and DPUs. The simplest three use 89 program steps (32bit words) to control circular buffers with two LMCs each. One uses 466 program steps to control eight LMCs that do some data reordering. Two others are used to control 55 DPUs and use 74 program steps. The control bus is only used once to transfer all the program steps to the ACs. If, however, the image size changes then all program steps need to be re-transferred, this is done automatically by the application software. For this application the six parameter bus have a peak data-rate of just 192 words per second. Again, this is much lower than the data-rate that the bus was designed for. The extremely low utilization of the parameter bus proves that the architecture is able to implement complex algorithms by using high data-rates in local connections, and low data rates in global control paths. And if for some reason higher rates are required, the system has enough headroom to meet them.

The FPGA with the most ACs and LMCs, the motion estimation and compensation FPGA, uses 15,7% of its used area for LMCs, ACs and parameter-bus. On the DWT FPGAs that number falls to 6%. The overhead of the weak-programmability is however lower because some of that area is for the address generators that would have to be present anyways.

A colleague did an implementation of the same algorithm in a Morpheus chip [103, 123]. A direct comparison is hard to do because Morpheus is a non-optimized prototype, however the FPGA area is smaller and the number of frames per second is bigger.

6.4. Summary and Conclusion

The chapter extensively covered the implementation details of a film grain noise reduction algorithm using the FlexWAFE architecture on the FlexFilm hardware board. It detailed firmware on all the four FlexFilm board FPGAs. The I/O FPGA contains a PCIe endpoint for communication with a PC and a control-bus bridge to configure the other FPGAs. All FPGAs are interconnected via point-to-point high-speed links and have 512MiB local SDRAM memory. Another FPGA implements the bidirectional motion estimation and compensation and the last two implement discrete wavelet transformations and noise filtering. Two DWT algorithm implementations were done, one based on FIR filters the other one on lifting filters. The FIR filter implementation required two FPGAs, the lifting implementation had a rounding non-linearity but required only a little over half an FPGA.

Weak-programmability was used whenever possible to provide functional flexibility without sacrificing too much area and with no performance penalty. LMCs were reused many times in the example application, and their function was changed by the local controllers. However, the application did not fully exploit the peak performance of the local controllers, they were designed for single clock cycle function switching between 16 functions at 125MHz (flexwafe board) and the application required only around 50 switches between at most 8 functions per

second. The architecture has the potential to cope with even more complex algorithms with more switches per second. This is a key advantage of the architecture, the data pipelines work always at full speed and do not stall because of the control pipelines. Control is distributed such that global control signals have very humble data rate requirements and therefore, no stalls occur. Furthermore the application benefited the LMC/LC flexibility, and they speed up development due to design reuse.

The noise reduction shrinkage filter DPUs also benefited from weak-programmability, it allowed the application to change the filter function without dynamically reconfiguring the FPGAs and used only 5 slices per data bit.

FlexWAFE contributed to the successful implementation of the algorithm in the flexfilm board. The noise reduction algorithm, although complex, did not exhaust the performance that FlexWAFE can provide.

7. Summary, Conclusion and Outlook

7.1. Brief overview

Chapter one provided an introduction to digital film processing and introduced the research projects that lead to the creation of the FlexWAFE. Chapter two gave an overview of other processing architectures and platforms for digital film. Some of their shortcomings were presented as motivation for the FlexWAFE development. Chapter three presented a bottom-up description of the flexwafe compositional blocks. It presented the data-path, image processing blocks, the data transport and reordering blocks and the control blocks. Chapter four explained the control hierarchy and how the system can be configured at synthesis and (re-)programmed at run-time. The simplicity of the control structures shown has benefits on both chip area and speed. Chapter five presented an example on how to configure and program a simple application. Here a step by step description on how to use the framework was also given. Chapter six described a complex noise reduction application implemented using the FlexWAFE. Many implementation details were given, and the results shown that the architecture is very suitable for this kind of applications.

7.2. Summary

This thesis introduced FlexWAFE, an FPGA-based, reconfigurable architecture and framework for digital film processing.

Image processing in general and digital film in particular have requirements of billions of operations per second and data rates in the range of terabits per second. This huge amount of data is due to the high resolution nature of film images and the need for processing at speeds exceeding 24 images per second.

These demanding requirements require a carefully planned design of the processing data-paths, data storage and global control. The FlexWAFE architecture provides an hardware proven library of blocks that solve many of the design issues, and the FlexWAFE framework provides automation to the design flow.

The communication part of the architecture (board-to-board, chip-to-chip, and external DDR-SDRAM communication) were the subject of [49]. This thesis was motivated mostly by:

- Performance of other solutions is not high enough.
- Design engineer productivity using traditional *functionally static HDL from scratch* methods is too low.

In order to address this, this thesis has focused on (in order of importance):

1. Maximize hardware reuse, while minimizing configuration time. FlexWAFE combines static configuration with dynamic weak programmability. This is a compromise between a pure static configuration and dynamic partial reconfiguration, it only requires a small area overhead for the control, allows dynamic function changes and avoids the high reconfiguration time penalty. Weak programmability saves mainly area, because blocks can perform more than one function (n-to-1 mapping, block reuse using parameters) instead of the traditional 1-to-1 mapping, and/or time because a FPGA reconfiguration would be slower than weak programmability (see chapter 4).
2. Unified interfaces for simpler compositional approach: back-pressure capable point-to-point stream interfaces for image data, memory-mapped bus interface with dedicated point-to-point *done* signals for control (see section 3.2).
3. Coding methodology/rules to standardize signal naming and interfaces, configurable using VHDL generics (see section 3.1).
4. High performance IP library that is portable across different FPGA families and generations (see section 5.4).
5. Reduce coding effort via reuse. For the FlexWAFE use case we developed a library of signal processing blocks specialized for film grain noise reduction (see section 6.2). However, many of the blocks can be reused in other applications due to the flexibility that the weak programmability brings.
6. Develop a semi-automated framework tool-chain that simplifies simulation, test and validation (see chapter 5).

In this thesis, we have presented the FlexWAFE architecture and framework for real-time digital film processing. Several applications where implemented using it on several hardware platforms. The most advanced platform used was the FlexFilm PCIe extension board with four FPGAs in a PC-based environment. Using this board the FlexWAFE IP core library and framework successfully achieved beyond 200 Gops/s and beyond 1 TBit/s internal data rate with a complex noise reduction algorithm.

7.3. Outlook

There are some open points that we feel should be addressed, but that we did not have the time to do, namely:

- When using bigger FPGAs replace the AC with a *soft-core* processor, it will provide greater flexibility, at the cost of complexity. This will allow to increase the flexibility of the control flow, allowing it to cover more use cases. To do so use a MIPS, or a Micro Blaze core, these are powerful yet simple cores.
- Extend the functionality of the local controllers, support more instructions and more program steps (newer Xilinx chips provide twice more memory at the LUT level, so it will not need any extra resources to double the amount of local controller memory). This allows more complex control to be implemented here, instead of in the AC.

Glossary and Acronyms

AC: Algorithm Controller,

central controller for a set of LMC(s) and DPU(s) that execute an algorithm vi, 25, 38–40, 43–50, 54, 56, 58, 60–62, 79, 93, 98, 108

ALU: Arithmetic Logic Unit,

an unit that performs arithmetic operations on one or more operands 17–19

API: Application Programming Interface,

a set of documented functions that one can use to interface to a library or device 54, 65

ASIC: Application Specific Integrated Circuit,

an integrated circuit designed for a certain application 3, 4, 23, 37

ASIP: Application-Specific Instruction-set Processor,

a processor with an instruction set tailored to a specific application 24

BMBF: Bundesministerium für Bildung und Foschung,

(German) Federal Office of Education and Research 1, 6

bpp: bits per pixel,

amount of bits used to represent the color or luminosity of an image pixel 65, 76, 112

causality

the output(s) and internal state(s) of a causal system depend on the current and past input value(s). In a non-causal system they also depend on future input values. 85

CGI: Computer Generated Images,

images produced by a computing device of any sort 3

CLB: configurable logic block,

basic component of Xilinx Virtex-II FPGAs containing 4 slices 74

CMC: Central Memory Controller,

SDRAM controller IP-core for the FlexFilm project 36, 37, 53, 59, 66, 72, 74, 77–79, 93, 108

CPU: Central Processing Unit,

a processor designed to perform diverse tasks and general computing 17–20, 24, 45, 64

CRT: Cathode Ray Tube,

a glass tube where beams of light produced in it's cathode are deflected to produce images on it's flat surface (anode) 3

DDR: double data rate,

synchronous transmission of data at twice the clock rate by using the rising and falling clock edge. 36, 100

DDR-SDRAM: Double Data Rate SDRAM,

SDRAM using DDR transmission to increase bandwidth 64, 103

DDWT: direct DWT,

direct discrete wavelet transformation, often just referred as DWT 85, 91, 92, 94, 114, 115

device driver

hardware-dependent software linked with the operating system, allowing user-programs to access the hardware via the OS that in turn uses the device-driver to communicate with the hardware 65

DI: Digital Intermediate,

a motion picture finishing process which involves digitizing a motion picture and manipulating its contents 2

DMA: direct memory access,

directly accessing a remote client's memory without processor intervention 19, 64–66, 93

DPU: Data Processing Unit,

flexible and programmable stream processing FPGA IP core (e.g. filter) without address generation, part of FlexWAFE IP library vi, ix, 25, 30–32, 35, 39, 40, 44, 45, 48–50, 53–56, 58, 62, 82

DPX: Digital Picture Exchange,

a common file format for digital intermediate and visual effects work and is an ANSI/SMPTE standard (268M-2003) 65

DSP: Digital Signal Processor,

a processor specialized for digital signal processing 3, 4

DWT: Discrete Wavelet Transformation,

a transformation to a space where wavelets are the base functions. 22, 28, 31, 70, 77, 83, 91, 92, 94, 100, 117

FFT: Fast Fourier Transformation,

an efficient algorithm to compute the discrete Fourier transform (DFT) and its inverse 22

FIFO: first-in-first-out,

dataflow sequence in some buffers 28, 32, 35, 36, 109, 110

FIR: Finite Impulse Response,

typically a property of convolution based digital filters ix, 30, 31, 81, 83, 85, 86, 91, 92, 94, 111, 115

FlexWAFE: Flexible Weakly-programmable Advanced Film Engine,

set of FPGA IP cores allowing creation of stream processing datapaths ix, 12, 14, 21, 25–27, 30–32, 36, 38–40, 44–51, 53, 54, 58, 59, 61–63, 66, 67, 93, 95–98, 109, 117

FPGA: Field Programmable Gate Array,

user programmable logic chip 3, 4, 12, 19, 22, 23, 25, 30, 32, 35, 37, 39, 44–46, 49, 58, 61–63, 65, 66, 68, 73, 74, 77, 80, 83, 90, 93, 94, 97–99, 101, 103, 104

fps: frames per second,

amount of image frames processed, transfered or displayed in the time interval of one second 65, 66

GPU: Graphics Processing Unit,

a processor specially tailored for graphics processing 3, 17

HDL: Hardware Description Language,

programming language to describe digital hardware behavior 22, 23, 97, 104

HT: HyperTransport,

communication framework 64

IDWT: inverse DWT,

inverse discrete wavelet transformation 85, 91, 92, 94

IP core

Intellectual Property core, a reusable unit of logic, cell, or chip layout design developed and owned by an individual or company 8, 12, 66, 98

JTAG: Joint Test Action Group,

a standard test access port and boundary-scan architecture defined in IEEE1149.1 92

LC: Local Controller,

Local controller inside FlexWAFE component 39, 40, 45–50

LCD: Liquid Crystal Display,

a thin, flat electronic visual display that uses the light modulating properties of liquid crystals 3, 64

LMC: Local Memory with Controller,

flexible and programmable stream address generator(s) including local FPGA memory buffers for stream based processing, part of FlexWAFE IP architecture vi, 25, 32–36, 39, 40, 44, 45, 48–50, 53–56, 58, 62, 72, 74, 77, 93, 108

LOTR: Lord of the Rings,

the movie trilogy by Peter Jackson 2

LSB: Least Significant Bit,

the bit with the smallest binary weight, usually the one on the right 82

LUT: look-up-table,

logic element of Xilinx Virtex-II FPGAs, can be configured as function generator with 4 inputs and one output, as 16 x 1-bit RAM or as 16 x 1-bit deep dynamic shift register 40, 74, 103, 116

LVDS: low voltage differential signaling,

a transmission standard which uses differential (complementary) but low voltages on two twisted wires to improve signal quality at high speeds 37, 64

MC: Motion Compensation,

an algorithm which assembles a new image from parts (blocks) of a previous image and the block motion vectors (the inverse of ME) 75, 76, 80, 93

ME: Motion Estimation,

an algorithm which derives the movement of image parts (blocks) relative to a previous image 28, 67, 74, 75, 77, 80, 93, 102

MIMD: Multiple Instruction Multiple Data,

multiple processor instructions operate on multiple data sources 40

MSB: Most Significant Bit,

the bit with the biggest binary weight, usually the one on the left 56

MV: Motion Vector,

vector which describes the movement of an image part (block) 28, 67–70, 72, 76, 77, 79, 93

NR: Noise Reduction,

to remove noise from a signal 22, 47, 90, 94

opcode: operation code,

a binary word that describes one specific processor instruction 49, 56

OS: Operating System,

a software that manages the hardware and software of a computer 44, 64, 100

PCB: Printed Circuit Board,

a board that provides physical support and electrical connection for electronic components 46, 64

PCI: Peripheral Component Interconnect,

bus based communication architecture found in PCs 7, 64, 102

PCIe: PCI-Express,

network style communication architecture found in modern PCs, successor of PCI and PCI-X 7, 44, 58, 63–66, 93, 98

PCI-X: PCI-Extended,

extended PCI bus, used in server or high-end workstation PCs 7, 64, 102

PE: Processing Element,

an atomic unit that performs a specific processing task 70–74

PG: Processing Group,

a group of DPUs without backpressure between them, backpressure is provided by BP blocks at the output(s) of the PG 31

QDR: quad data rate,

synchronous transmission of data at quad clock rate. 37

RAID: Redundant Array of Independent Disks,

a system to increase hard-disk reliability and/or data throughput 65

RGB: Red Green Blue,

a technique to quantify a color by decomposing it in three components red, green and blue 1, 2, 4, 28, 30, 67, 71, 75, 83, 92

RIO: RapidIO,

network based communication framework 64

RISC: Reduced Instruction Set Computer,

a CPU design with simplified instructions that execute faster when compared to CISC instructions 17, 19, 49

RTL: register transfer level,

a synthesizable way to describe a digital logic circuit by defining registers and the boolean logic that interconnects them 32

SAD: Sum of Absolute Differences,

matching kernel for block matching 69, 70, 74–77, 79

SDF: Synchronous Data Flow,

a model of computation mainly used to describe signal processing tasks 26, 27, 30, 53

SDR: single data rate,

synchronous transmission of data at clock rate. 103

SDRAM: Synchronous Dynamic Random Access Memory,

modern, widely used DRAM technology running synchronous to a clock, describes a whole family: Single Data Rate SDRAM (SDR-SDRAM), DDR-SDRAM, DDR2-SDRAM, DDR3-SDRAM. 6, 7, 9, 20, 25, 35–37, 59, 62, 71, 77, 93, 100, 103, 108–110

SDR-SDRAM: Single Data Rate SDRAM,

original SDRAM using single data rate (SDR) data transmission, now used to distinguish from DDR-SDRAM. 103

SIMD: Single Instruction Multiple Data,

the same processor instruction operates on multiple data sources 31

slice

low-level component of an Xilinx Virtex-II FPGA containing the user-configurable logic (2 LUTs and 2 flipflops) 32

SoC: System-on-Chip,

a complete system implemented in a single chip using one or more integration technologies 4

SPE: Symmetrical Periodic Extension,

a method to extend finite signals in order to process them using methodologies for periodic signals 85, 86, 89

SRAM: Static Random Access Memory,

volatile, low-density, high cost, widely used RAM technology 20

systolic array

a network of processors which rhythmically compute and pass data through the system. Every processor regularly pumps data in and out, each time performing some short computation, so that a regular flow of data is kept up in the network. These systolic arrays enjoy simple and regular communication paths, and almost all processors used in the networks are identical [72] 71, 72

TDM: time division multiplexing,

a scheduling technique which arbitrates between flows using a fixed sequence. 25, 37, 66, 104

TDMA: time division multiple access,

see TDM 37

Verilog

a HDL language used to model electronic systems 22, 23

VHDL: Very High Speed Integrated Circuit HDL,

a HDL language used to model electronic systems 6, 9, 22, 23, 26, 28, 40, 45, 46, 48, 50, 53, 61, 98, 117

Virtex

a FPGA product name used by Xilinx, it currently has six sub-families: Virtex, Virtex-II (Pro), Virtex-4, Virtex-5, Virtex-6, Virtex-7 32, 99, 101, 103

VLIW: Very Large Instruction Word,

a CPU architecture designed to take advantage of instruction level parallelism 19, 40

Xilinx

one of the world's biggest FPGA manufacturer 19, 22, 32, 40, 43, 61, 104

XML: Extensible Markup Language,

a set of rules for encoding documents in machine-readable form 40, 44–49, 53, 54, 62, 117

YCbCr: Luminance, Blue-difference, Red-difference,

a technique to quantify a color by decomposing it in three components as defined by CIR 601 4, 30

A. Vocabulary and Notation

A.1. Binary prefixes

In order to distinguish between a base of 2^{10} and 10^3 the IEC-60027-2 [1] norm will be used, with "b" indicating "bits" and "B" indicating "bytes". See table A.1 for examples.

A.2. Rounding

Unless otherwise stated, values are always rounded to three significant digits, for example 1.234 to 1.23; 12,340 to 12,300; 0.001234 to 0.00123.

A.3. Bandwidth versus Data Rate

The term "bandwidth" means the difference between the upper and lower cutoff frequencies of, for example, a communication channel or a signal spectrum and is typically expressed in hertz. In the computer and networking literature, for example [54, 70, 62, 101] the term "bandwidth" is often incorrectly used as a synonym for "data rate" which is measured in bits per second, for example to designate a channel's throughput.

The reason for this usage is that according to Shannon [106] the maximum data rate or channel capacity in bit/s is proportional to the analogue bandwidth in hertz:

$$C = B \log_2 \left(1 + \frac{S}{N}\right)$$

Decimal		Binary	
Kb	$10^3 = 1,000$ bit	Kib	$2^{10} = 1,024$ bit
Mb	$10^6 = 1,000,000$ bit	Mib	$2^{20} = 1,048,576$ bit
Gb	$10^9 = 1,000,000,000$ bit	Gib	$2^{30} = 1,073,741,824$ bit
KB	$10^3 = 1,000$ byte	KiB	$2^{10} = 1,024$ byte
MB	$10^6 = 1,000,000$ byte	MiB	$2^{20} = 1,048,576$ byte
GB	$10^9 = 1,000,000,000$ byte	GiB	$2^{30} = 1,073,741,824$ byte

Table A.1.: binary prefixes

whereas C is the channel capacity expressed in bit/s, B is the channel bandwidth expressed in hertz, S is the total signal power over the bandwidth, measured in volt^2 or watt, and N is the total white Gaussian noise power over the bandwidth, measured in volt^2 or watt.

In this thesis the correct term "data rate" will be used.

B. Memory based circular buffers

In image processing systems it is very typical to store image frames in circular or ring buffers. They consist of a single fixed size buffer, divided in two or more equally sized segments. To access the segments, a combination of base address pointer + address pointer is used. The base address pointers point to the beginning of the segments and the address pointers are non-negative offsets within the segments, therefore are smaller than the segment size.

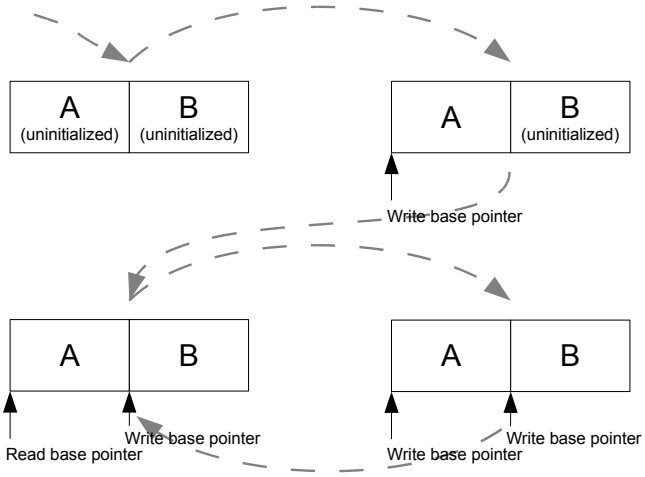


Figure B.1.: Double buffer - pointers and states

A double buffer has two segments called A and B. In the initial state (Figure B.1 top left), both segments are uninitialized, and therefore do not contain any valid data. Then an input image stream is written to segment A, it can be written in any order deemed necessary. After the input stream has been completely written, the base pointer of that stream switches to segment B, and a new stream can be again written using the same address pattern (Figure B.1 top right). This simplifies the system because the only thing that changes is the base address pointer. Now that segment A has been completely written to, an output stream can be read from it using any image pattern deemed necessary (Figure B.1 bottom left). It must not necessarily match the access pattern used to write it, and that can be of major importance for some algorithms. To be able to do so concurrently the double buffer needs to have two concurrent access ports. The state machine that controls the double buffer will then wait until both segment B has been completely written to and segment A read from. After that, it switches the base pointers (Figure B.1 bottom right) and it will allow write to segment A and reading from segment B. The last two states then alternate cyclically. Basically it only allows read opera-

tions from segments that have been completely written to, and write operations to segments that haves been completely read or are uninitialized.

This very same concept is used when more segments are used. Throughout this thesis two, four and eight segments where used in the several line buffers and LMC implementations.

In the motion estimation and motion compensation four segment circular buffers using external SDRAM memory were used (Figure 6.5 on page 74 and Figure 6.10 on page 78). This was necessary because one buffer was used to write the incoming current image, but the past three images where needed for the algorithms, hence four segments where required. The multi-ported CMC allowed pseudo-concurrent access to the circular buffers, the pointer address sequences where computed by LMC_S2C and LMC_C2S and the circular buffer states where controlled by the ACs and their software based programs. A detailed example of an SDRAM based double buffer is presented in 5.3.

C. Asynchronous FIFOs

Asynchronous FIFOs have been briefly introduced on Section 3.4.1. This appendix gives a more in depth explanation about the topic and the FlexWAFE implementation.

Asynchronous FIFOs have two clock domains. The write control signals and the data to be written are synchronous with one clock domain, the read signals and the read data are synchronous with another clock domain. The frequency of one of the clocks need not be a multiple from the other and clock jitter is allowed in both clocks. The challenge in these requirements is to correctly generate full and empty FIFO signals. These signals are required to control the read and write process in order to avoid buffer under-flows (a read attempt on an empty FIFO) and buffer overflows (a write attempt on a full FIFO).

Typically in RAM based FIFOs an incrementing write pointer and an incrementing read pointer are used and empty and full conditions are detected by comparing the difference between these two pointers. In fully synchronous FIFOs, binary counters are used to generate the two pointers. In asynchronous FIFOs, the binary codes are converted to gray code before comparing them. This avoids glitches thus avoiding false full and false empty detection [28]. We decided to directly use gray encoded counters instead of doing the binary to gray conversion, which increased performance and decreased hardware resources. As a side effect, the data is not stored in linear order (consecutive addresses) in RAM, but is stored in gray code order. That is however not an issue because it is also retrieved in that same order.

In some applications, data is delivered or consumed in a burst oriented fashion. A burst simply means that n consecutive data elements are transmitted back to back without interruption during n clock cycles. An example of such an interface is presented in section 3.5. For a normal FIFO such a data transfer can pose a problem: once a burst data transfer is started, there is no way to interrupt it and therefore the FIFO might run empty or full if at the moment when the burst started the FIFO was almost empty or almost full. To solve this issue, some FIFO implementations provide extra flags for *almost empty* when the number of data tokens in the FIFO is smaller than m and *almost full* when the number of data tokens in the FIFO is greater than k. Where m and k are configurable natural numbers, typically set so that $m = n$ and $k = FIFOdepth - n$. Our implementation also provides this feature.

For interfacing with external SDRAM or other types of relatively high latency peripherals a more intelligent data-flow management is required. If for example a FIFO depth of 512 is used, with a write burst length of 16, a read length of 1 and a worst case latency of 1000 clock cycles then it might happen that the FIFO runs almost full, the write operations will stop, the last write operation hits the worst-case scenario and the read requests continue. The last write operation will only complete after 1000 clock cycles, by that time the FIFO will most probably be empty because it holds less than 1000 tokens. Empty FIFOs are to be avoided because they stall the entire processing data-paths, so one solution is to increase the FIFO

depth to be slightly bigger than the worst case latency. But that might not be possible in situations where the latency is too high and/or the amount of memory resources available is limited.

Now consider the example above but with a starting condition of almost empty and a read data rate of 1 token every 10 clock cycles. It might happen that the sate machine that controls the write operations initiates too many requests (>512/16) because the FIFO is not almost full and the write operations do not start immediately, but after some latency. By the time the data starts to arrive, and assuming that the data arrives burst after burst without interruption, the FIFO is still not almost full and as such some more write requests are initiated, but these will not fit in the fast filling FIFO. To avoid such scenarios a look-ahead counter mechanism must be implemented that takes not only read/writes into account but also initiated burst request operations that will lead to future read/write operations.

In order to do so our FIFO implementation contains *read burst request* and *write burst request* inputs, configurable burst length, as well as *will be empty* and *will be full* outputs. Using these signals simplifies the design of the state machines that schedule the burst oriented requests. The advantage of doing so is that the read side of the FIFO and all it's controlling logic are kept in one clock domain while the write side and it's control logic are kept in another. All clock crossing logic is contained in the asynchronous FIFO itself and must only be tested and validated once.

To decrease the average latency when accessing external SDRAM it is necessary to perform requests before they are actually required. This will allow the schedulers inside the SDRAM controller to choose and optimized access sequence and therefore decrease average latency. This technique is commonly referred as pre-fetching and was proved very effective in accessing external SDRAM as described in section 3.5 on page 36. The address generators presented in Section 3.4.2 on page 32 use the *will be empty* and *will be full* outputs of the FIFO to easily pre-fetch as many data tokens as possible to benefit from the reduced latency that such technique achieves.

D. Convolution

A common way of filtering a discrete digital signal is to use the mathematical convolution operation between the signal and the filter impulse response [89]. Convolution is used throughout this thesis, and it is assumed that the reader is familiar with it. This chapter will explain what is convolution, for the readers that are not familiar with it, and how the dynamic range of the intermediate results are calculated. The dynamic range of the results are particularly interesting for the hardware implementation because it directly affects the number of bits required to represent them.

D.1. Definition and properties

Given a discrete signal $x[n] \in \mathbb{Z} \wedge n \in \mathbb{Z}$ and a filter impulse response $h[n] \in \mathbb{Z} \wedge n \in \mathbb{Z}$ then their convolution $y[n]$ is defined as:

$$y[n] = x[n] * h[n] \stackrel{def}{=} \sum_{j=-\infty}^{+\infty} x[j]h[n-j] \tag{D.1}$$

This is a very straightforward operation especially for FIR filters. On this thesis we focused on digital image processing using 2D wavelets of finite length. Therefore both the signal and the filter impulse response have finite length but unlike in Equation D.1 are two-dimensional signals:

$$y[m,n] = x[m,n] * h[m,n] \stackrel{def}{=} \sum_{j=-\infty}^{+\infty} \sum_{i=-\infty}^{+\infty} x[i,j]h[m-i,n-j] \tag{D.2}$$

The used wavelet filters (5/3 LeGall) are separable to (Mx1) and (1xN) 1D vectors ($M, N \in \mathbb{N}$), therefore by definition, h_1 and h_2 orthogonal functions exist such that:

$$h[m,n] \stackrel{def}{=} h_1[m]h_2[n] \tag{D.3}$$

Since convolution is commutative ($x[n] * h[n] = h[n] * x[n]$), we can swap the order of convolution in a first step, then apply Equation D.3, and the commutativity propriety to get:

$$
\begin{aligned}
y[m,n] = x[m,n] * h[m,n] &= \sum_{j=-\infty}^{+\infty} \sum_{i=-\infty}^{+\infty} h[i,j]x[m-i,n-j] \\
&= \sum_{j=-\infty}^{+\infty} \sum_{i=-\infty}^{+\infty} h_1[i]h_2[j]x[m-i,n-j] \\
&= \sum_{j=-\infty}^{+\infty} h_2[j] \left(\sum_{i=-\infty}^{+\infty} h_1[i]x[m-i,n-j] \right)
\end{aligned}
$$

Comparing the result above, and Equation D.1 it can be seen that it is convolving input and $h_1[m]$ and then convolving the result with $h_2[n]$. Therefore, the separable 2D convolution is performing two 1D convolutions, one in the horizontal and one in the vertical direction. And, convolution is associative [89], it does not matter which direction is performed first. Therefore, one can convolve with $h2[n]$ first then convolve with $h1[m]$ later.

$$
\begin{aligned}
y[m,n] = (h_1[m]h_2[n]) * x[m,n] &= h_2[n] * (h_1[m] * x[m,n]) \\
&= h_1[m] * (h_2[n] * x[m,n])
\end{aligned}
$$

In our filters $h_1[m] = (h_2[n])^T$ and the filter lengths are the same $(M = N)$ so we can drop the subscripts and use the same filter $h[\,]$ in both directions.

D.2. Dynamic range

The image stream signal is discrete (set of pixels) and its range is also finite and quantized to fit the interval achievable with bits per pixel (bpp). To avoid overflows and underflows, it is required to use extra guard bits when computing the results of the filtering operations. The amount of guard bits for one filter operation is calculated by:

$$
\textit{guard bits} = \left\lceil log_2 \left(\sum_{m=-\infty}^{+\infty} |h[m]| \right) \right\rceil \tag{D.4}
$$

On a first, naive, and over-conservative attempt to calculate the guard bits, Equation D.4 was used recursively in every filter. The bitwidth at the output of the first filter (k=1) was bpp plus guard bits of the first filter. The bitwidth at the output of the second filter (k=2) was the output bitwidth of the first filter plus guard bits of the second filter, and so on.

$$
\textit{bits}[k] = \textit{bits}[k-1] + \left\lceil log_2 \left(\sum_{m=-\infty}^{+\infty} |h[m]| \right) \right\rceil \tag{D.5}
$$

This first approach is over-conservative because it does not take into consideration that the impulse response of h^k is actually the convolution of the impulse response of the previous filters (h^{k-1} and h^{k-2}). Its results are shown on Figure D.1.

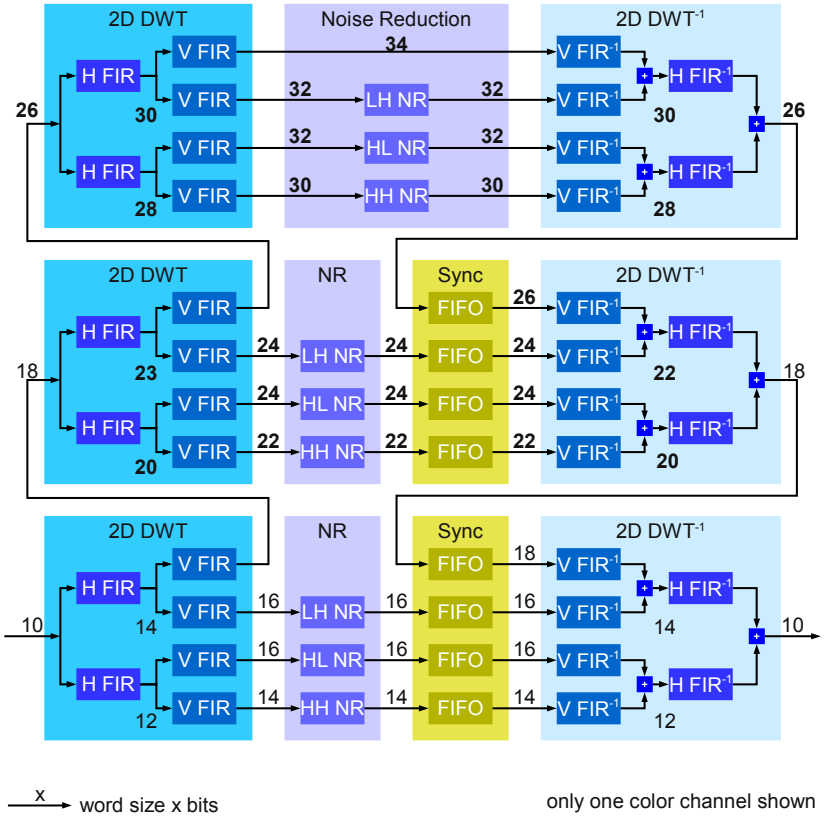


Figure D.1.: Lifting based direct 2D three level DWT (left side), filtering and buffering in the wavelet space (center), followed by inverse 2D three level DWT (right side) using integer filter coefficients and **ignoring** filter cascading effects on the streams dynamic range. Over-conservative bitwidths are marked in bold

D.2.1. Integer coefficients

It should also be taken into account that although the horizontal filters are interleaved with the vertical filters, their effect is independent in regard to the impulse response. In other words, the impulse response of the third horizontal filter is the convolution of the first horizontal filter with the second horizontal filter; the vertical filters in between are not to be convoluted, instead their effect is to be considered as a scalar multiplication.

Using equations 6.15 and 6.16, h_L^1 and h_H^1 (first level DDWT) can be defined as:

$$h_L^1 = [-1,\ 2,\ 6,\ 2,\ -1] \tag{D.6}$$
$$h_H^1 = [-1,\ 2,\ -1] \tag{D.7}$$

The impulse response h_L^2 and h_H^2 for the second level DDWT filters is defined (due to the up-sampling by two) as:

$$h_L^2 = [-1,\ 0,\ 2,\ 0,\ 6,\ 0,\ 2,\ 0,\ -1] \tag{D.8}$$
$$h_H^2 = [-1,\ 0,\ 2,\ 0,\ -1] \tag{D.9}$$

The impulse response h_L^3 and h_H^3 for the third level DDWT filters is defined (due to the up-sampling by two) as:

$$h_L^3 = [-1,\ 0,\ 0,\ 0,\ 2,\ 0,\ 0,\ 0,\ 6,\ 0,\ 0,\ 0,\ 2,\ 0,\ 0,\ 0,\ -1] \tag{D.10}$$
$$h_H^3 = [-1,\ 0,\ 0,\ 0,\ 2,\ 0,\ 0,\ 0,\ -1] \tag{D.11}$$

Assuming $x[n]$ input values in the interval $[-2^{bpp-1},\ 2^{bpp-1}-1]$ the dynamic ranges of the first DDWT level are:

$$bits_L = \left\lceil log_2 \left(2^{BPP-1} \sum_{n=-2}^{+2} \left| h_L^1[n] \right| \right) \right\rceil + 1 \tag{D.12}$$

$$bits_H = \left\lceil log_2 \left(2^{BPP-1} \sum_{n=-1}^{+1} \left| h_H^1[n] \right| \right) \right\rceil + 1 \tag{D.13}$$

$$bits_{LL} = \left\lceil log_2 \left(2^{BPP-1} \sum_{m=-2}^{+2} \sum_{n=-2}^{+2} \left| h_L^1[m]^T * h_L^1[n] \right| \right) \right\rceil + 1 \tag{D.14}$$

$$bits_{LH} = \left\lceil log_2 \left(2^{BPP-1} \sum_{m=-1}^{+1} \sum_{n=-2}^{+2} \left| h_L^1[m]^T * h_H^1[n] \right| \right) \right\rceil + 1 \tag{D.15}$$

$$bits_{HL} = \left\lceil log_2 \left(2^{BPP-1} \sum_{m=-2}^{+2} \sum_{n=-1}^{+1} \left| h_H^1[m]^T * h_L^1[n] \right| \right) \right\rceil + 1 \tag{D.16}$$

$$bits_{HH} = \left\lceil log_2 \left(2^{BPP-1} \sum_{m=-1}^{+1} \sum_{n=-1}^{+1} \left| h_H^1[m]^T * h_H^1[n] \right| \right) \right\rceil + 1 \tag{D.17}$$

The dynamic ranges of the second DDWT level are:

$$bits_{LLL} = \left\lceil log_2 \left(2^{BPP-1} \sum_{m=-2}^{+2} \sum_{n=-6}^{+6} \left| h_L^1[m]^T * h_L^1[n] * h_L^2[n] \right| \right) \right\rceil + 1 \tag{D.18}$$

$$bits_{LLH} = \left\lceil log_2 \left(2^{BPP-1} \sum_{m=-2}^{+2} \sum_{n=-4}^{+4} \left| h_L^1[m]^T * h_L^1[n] * h_H^2[n] \right| \right) \right\rceil + 1 \tag{D.19}$$

$$bits_{LLLL} = \left\lceil log_2 \left(2^{BPP-1} \sum_{m=-6}^{+6} \sum_{n=-6}^{+6} \left| h_L^1[m]^T * h_L^1[n] * h_L^2[m]^T * h_L^2[n] \right| \right) \right\rceil + 1 \tag{D.20}$$

$$bits_{LLLH} = \left\lceil log_2 \left(2^{BPP-1} \sum_{m=-4}^{+4} \sum_{n=-6}^{+6} \left| h_L^1[m]^T * h_L^1[n] * h_L^2[m]^T * h_H^2[n] \right| \right) \right\rceil + 1 \tag{D.21}$$

$$bits_{LLHL} = \left\lceil log_2 \left(2^{BPP-1} \sum_{m=-6}^{+6} \sum_{n=-4}^{+4} \left| h_L^1[m]^T * h_L^1[n] * h_H^2[m]^T * h_L^2[n] \right| \right) \right\rceil + 1 \tag{D.22}$$

$$bits_{LLHH} = \left\lceil log_2 \left(2^{BPP-1} \sum_{m=-4}^{+4} \sum_{n=-4}^{+4} \left| h_L^1[m]^T * h_L^1[n] * h_H^2[m]^T * h_H^2[n] \right| \right) \right\rceil + 1 \tag{D.23}$$

To simplify the following equations we combine the impulse response of the first and second level DDWT filters and define:

$$h_L^{12}[m,n] = h_L^1[m]^T * h_L^1[n] * h_L^2[m]^T * h_L^2[n] \tag{D.24}$$

The dynamic ranges of the third DDWT level are:

$$bits_{LLLLL} = \left\lceil log_2 \left(2^{BPP-1} \sum_{m=-6}^{+6} \sum_{n=-14}^{+14} \left| h_L^{12}[m,n] * h_L^3[n] \right| \right) \right\rceil + 1 \tag{D.25}$$

$$bits_{LLLLH} = \left\lceil log_2 \left(2^{BPP-1} \sum_{m=-6}^{+6} \sum_{n=-10}^{+10} \left| h_L^{12}[m,n] * h_H^3[n] \right| \right) \right\rceil + 1 \tag{D.26}$$

$$bits_{LLLLLL} = \left\lceil log_2 \left(2^{BPP-1} \sum_{m=-14}^{+14} \sum_{n=-14}^{+14} \left| h_L^{12}[m,n] * h_L^3[m]^T * h_L^3[n] \right| \right) \right\rceil + 1 \tag{D.27}$$

$$bits_{LLLLLH} = \left\lceil log_2 \left(2^{BPP-1} \sum_{m=-10}^{+10} \sum_{n=-14}^{+14} \left| h_L^{12}[m,n] * h_L^3[m]^T * h_H^3[n] \right| \right) \right\rceil + 1 \tag{D.28}$$

$$bits_{LLLLHL} = \left\lceil log_2 \left(2^{BPP-1} \sum_{m=-14}^{+14} \sum_{n=-10}^{+10} \left| h_L^{12}[m,n] * h_H^3[m]^T * h_L^3[n] \right| \right) \right\rceil + 1 \tag{D.29}$$

$$bits_{LLLLHH} = \left\lceil log_2 \left(2^{BPP-1} \sum_{m=-10}^{+10} \sum_{n=-10}^{+10} \left| h_L^{12}[m,n] * h_H^3[m]^T * h_H^3[n] \right| \right) \right\rceil + 1 \tag{D.30}$$

The results of all these equations for bpp=10 are shown on Figure 6.13 on page 86 for a FIR based implementation with integer coefficients. And on Figure 6.18 on page 91 for a

lifting based implementation with fractional coefficients from Table 6.2 on page 90. One other difference is that the FIR implementation does horizontal filtering before the vertical filtering and the lifting implementation does the opposite. But due to the commutativity property of the convolution, this implementation difference has no (mathematical) impact on the results.

D.2.2. Fractional coefficients

For the lifting implementation that maps integers into integers using fractional filter coefficients as explained on Section 6.2.3 on page 82 equations D.6, D.7, D.8, D.9, D.10 and D.11 are respectively redefined as:

$$h_L^1 = [-1/8,\ 1/4,\ 3/4,\ 1/4,\ -1/8] \tag{D.31}$$
$$h_H^1 = [-1/2,\ 1,\ -1/2] \tag{D.32}$$
$$h_L^2 = [-1/8,\ 0,\ 1/4,\ 0,\ 3/4,\ 0,\ 1/4,\ 0,\ -1/8] \tag{D.33}$$
$$h_H^2 = [-1/2,\ 0,\ 1,\ 0,\ -1/2] \tag{D.34}$$
$$h_L^3 = [-1/8,\ 0,\ 0,\ 0,\ 1/4,\ 0,\ 0,\ 0,\ 3/4,\ 0,\ 0,\ 0,\ 1/4,\ 0,\ 0,\ 0,\ -1/8] \tag{D.35}$$
$$h_H^3 = [-1/2,\ 0,\ 0,\ 0,\ 1,\ 0,\ 0,\ 0,\ -1/2] \tag{D.36}$$

These equations can be used together with equations D.12 through D.30 to calculate the dynamic range of the fractional lifting implementation as seen on Figure 6.18 on page 91.

As can be seen on that figure, using fractional filter coefficients can save a lot of logic resources, less registers are needed to store the intermediate results and less LUTs are needed to compute them. This translates into less FPGA area, less routing congestion and higher F_{max} (maximum achievable operating frequency).

E. Publications

In the context of this thesis, several publications were accomplished and are summarized here by order of publication. As first author:

An Image Processor for Digital Film; Amilcar do Carmo Lucas and Rolf Ernst; In IEEE Application-Specific Systems, Architectures, and Processors (ASAP); 2005, pp 219–224, July 2005 [37].

This paper introduces the FlexWAFE architecture and the first steps of the DWT hardware implementation.

A reconfigurable HW/SW platform for computation intensive high-resolution real-time digital film applications; Amilcar do Carmo Lucas, Sven Heithecker, Peter Rüffer, Rolf Ernst, Holger Rückert, Gerhard Wischermann, Karin Gebel, Reinhard Fach, Wolfgang Hunther, Stefen Eichner, Gunter Scheller; Proceedings of the Conference on Design, Automation and Test in Europe (DATE); 2006, Munich (Germany) [39].

This paper covers the whole FlexFilm architecture, with a focus on the implementation of the noise reduction application. This paper won a *system design record award*.

FlexWAFE - A high-end real-time stream processing library for FPGAs; Amilcar do Carmo Lucas and Sven Heithecker and Rolf Ernst; Design Automation Conference (DAC); 2007, New York (NY, USA) [38].

This paper covered the VHDL model configuration using XML.

Application Development with the FlexWAFE Real-Time Stream Processing Architecture for FPGAs; Amilcar do Carmo Lucas and Henning Sahlbach and Sean Whitty and Sven Heithecker and Rolf Ernst; ACM Transactions on Embedded Computing Systems; 2009 [40].

This journal paper explains how to use the developed architecture and presents some example applications.

As co-author:

A Mixed QoS SDRAM Controller for FPGA-Based High-End Image Processing; Sven Heithecker, Amilcar do Carmo Lucas, Rolf Ernst; Workshop on Signal Processing Systems Design and Implementation; 2003, Seoul (South Korea) [50].

In this paper two architecture variants for the FlexFilm SDRAM Controller were discussed.

FlexFilm - an Image Processor for Digital Film Processing; Sven Heithecker, Amilcar do Carmo Lucas, Rolf Ernst; Workshop on Dynamically Reconfigurable Architectures; 2006, Dagstuhl (Germany) [51].

This paper gives an overview about the complete FlexFilm architecture.

A High-End Real-Time Digital Film Processing Reconfigurable Platform; Sven Heithecker, Amilcar do Carmo Lucas, Rolf Ernst; EURASIP Journal on Embedded Systems, Special Issue on Dynamically Reconfigurable Architectures [52].

Comprehensive journal article about the FlexFilm project.

FlexWAFE: FPGA-basierte Bildverarbeitung für digitales Kino; Henning Sahlbach, Sean Whitty, Amilcar do Carmo Lucas, Rolf Ernst; FKT 2009 [104].

Presents the applications developed especially for digital cinema.

Bibliography

[1] *IEC 60027-2: Letter symbols to be used in electrical technology - Part 2: Telecommunications and electronics*, 3rd ed. IEC, August 2005.

[2] "NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi™," Whitepaper, 2009.

[3] Achronix, "Speedster® FPGA Family," Product Brief, 2009.

[4] Agilent Ptolemy. Online: http://eesof.tm.agilent.com/products/e8823a-new.html

[5] M. Aitken, G. Butler, D. Lemmon, E. Saindon, D. Peters, and G. Williams, "The Lord of the Rings: the visual effects that brought middle earth to the screen," in *ACM SIGGRAPH 2004 Course Notes*, ser. SIGGRAPH '04. New York, NY, USA: ACM, 2004. Online: http://doi.acm.org/10.1145/1103900.1103911

[6] Altera Corporation. Online: http://www.altera.com

[7] Apple Inc. Online: http://www.apple.com

[8] Arri. Online: http://www.arri.com

[9] Aspex Semiconductor. Online: http://www.aspex-semi.com

[10] Autodesk. Online: http://usa.autodesk.com

[11] R. Baxter, S. Booth, M. Bull, G. Cawood, K. D'Mellow, X. Guo, M. Parsons, J. Perry, A. Simpson, and A. Trew, "High-Performance Reconfigurable Computing - the View from Edinburgh," *Adaptive Hardware and Systems, NASA/ESA Conference on*, vol. 0, pp. 273–279, 2007.

[12] K. J. Bean, A. Krikelis, I. P. Jalowiecki, D. Bean, R. Bishop, M. Facey, D. Boughton, S. Murphy, M. Whitaker, K. Lane, and U. U. Ph, "A Programmable Processor with 4096 Processing Units for Media Applications," in *in Proc. of the IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, 2001, pp. 937–940.

[13] L. Benini, M. Lombardi, M. Milano, and M. Ruggiero, "A Constraint Programming Approach for Allocation and Scheduling on the CELL Broadband Engine," in *Principles and Practice of Constraint Programming*, ser. Lecture Notes in Computer Science, P. Stuckey, Ed. Springer Berlin / Heidelberg, 2008, vol. 5202, pp. 21–35, 10.1007/978-3-540-85958-1_2. Online: http://dx.doi.org/10.1007/978-3-540-85958-1_2

[14] Berkeley Design Technology, Inc., "An Independent Evaluation of: The AutoESL AutoPilot High-Level Synthesis Tool," p. 17, 2010.

[15] C. M. Brislawn, "Classification of nonexpansive symmetric extension transforms for multirate filter banks," in *Applied and Computational Harmonic Analisys*, vol. 3, 1996, pp. 337–357.

[16] S. Brown, D. Tracy, K. Kho, and T. Margolis, "NEXT Generation Digital MEDIA," *USCD center for hybrid multicore productivity research*, October 2009.

[17] S. Brüne, "Standard and High defenition SDI video I/O interface using the Virtex FPGAs of the PY1013 board," Studienarbeit, IDA, TU-Braunschweig, August 2005, STA3341.

[18] J. Casazza, "First the tick, now the tock: Intel microarchitecture (nehalem)," *Intel Xeon Processor 3500 and 550 series*, 2009.

[19] Celco. Online: http://www.celco.com

[20] S.-C. Cheng and H.-M. Hang, "A Comparison of Block-Matching Algorithms Mapped to Systolic-Array Implementation," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 7, pp. 741–757, 1997.

[21] CineGrid a non-profit organization. Online: www.cinegrid.org

[22] Cinepaint. Online: http://www.cinepaint.org

[23] Cintel. Online: http://www.cintel.co.uk

[24] B. Cohen, *VHDL Coding Styles and Methodologies*. Norwell, MA, USA: Kluwer Academic Publishers, 1995.

[25] Compaan Design BV, "Compaan Xilinx FPGA Code Generator," Compaan Design BV, Tech. Rep., 2010.

[26] P. Conway and B. Hughes, "The amd opteron northbridge architecture," *IEEE Micro*, vol. 27, pp. 10–21, March 2007. Online: http://portal.acm.org/citation.cfm?id=1308421.1308521

[27] J. Corbet, G. Kroah-Hartman, and A. Rubini, *Linux Device Drivers*, 3rd ed. O'Reilly, February 2005.

[28] C. E. Cummings and P. Alfke, "Simulation and synthesis techniques for asynchronous fifo design with asynchronous pointer comparisons," *Synopsys Users Group*, p. 18, April 2002.

[29] B. Dally, "Parallel Processing Simplified Storm-1 SP16HP: A 112 GMAC, C-programmable Media and Signal Processor," in *Microprocessor Forum*, 2007.

[30] B. Dally, "Stream Processing: Enabling the new generation of easy to use, high-performance DSPs," *SPI*, p. 11, June 2008.

[31] Dalsa. Online: http://www.dalsa.com

[32] DaVinci. (2008) Homepage. Online: http://geniusofdavinci.com

[33] Definity. Online: http://www.definity35mm.com

[34] Department of Electronic Circuits and Systems, working group DigitalImage and Video Processing, Ilmenau University of Technology. Online: http://wcms1.rz.tu-ilmenau.de/fakei/Arbeitsgruppe-Digita.1378.0.html?&L=1

[35] digitalvision. Online: http://www.digitalvision.se

[36] Q. Dinh, D. Chen, and M. D. F. Wong, "Efficient asip design for configurable processors with fine-grained resource sharing," in *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, ser.

FPGA '08. New York, NY, USA: ACM, 2008, pp. 99–106. Online: http://doi.acm.org/10.1145/1344671.1344687

[37] A. do Carmo Lucas and R. Ernst, "An Image Processor for Digital Film," in *IEEE ASAP*, July 2005. Online: http://www.ece.uvic.ca/asap2005/

[38] A. do Carmo Lucas, S. Heithecker, and R. Ernst, "FlexWAFE - A high-end real-time stream processing library for FPGAs," in *DAC '07: Proceedings of the 44th Annual conference on Design Automation*. New York, NY, USA: ACM Press, 2007, pp. 916–921. Online: http://www2.dac.com/data2/44th/44acceptedpapers.nsf/0c4c09c6ffa905c487256b7b007afb72/811edf2d2db321c9872572a000471d72?OpenDocument

[39] A. do Carmo Lucas, S. Heithecker, P. Rüffer, R. Ernst, H. Rückert, G. Wischermann, K. Gebel, R. Fach, W. Hunther, S. Eichner, and G. Scheller, "A reconfigurable HW/SW platform for computation intensive high-resolution real-time digital film applications," in *IEEE DATE*, March 2006, pp. 194–199.

[40] A. do Carmo Lucas, H. Sahlbach, S. Whitty, S. Heithecker, and R. Ernst, "Application development with the FlexWAFE real-time stream processing architecture for FPGAs," *ACM Transactions on Embedded Computing Systems, Special Issue on Configuring Algorithms, Processes and Architecture (CAPA)*, vol. 9, no. 1, p. 23, October 2009. Online: http://doi.acm.org/10.1145/1596532.1596536

[41] S. Dutta, R. Jensen, and A. Rieckmann, "Viper: A multiprocessor SoC for advanced set-top box and digital TV systems," in *IEEE Design and Test of Computers, Sip*, Oct 2001, pp. 21–31.

[42] S. Eichner, G. Scheller, U. Wessely, H. Rückert, and R. Hedtke, "Motion compensated spatial-temporal reduction of film grain noise in the wavelet domain," in *SMPTE Technical Conference, New York*, 2005.

[43] S. Eichner, G. Scheller, and U. Wessely, "Wavelet-temporal basierende Rauschreduktion von Filmsequenzen," in *21. Jahrestagung der FKTG, Koblenz*, May 2004. Online: http://www.flexfilm.org/publications_local/ESW04:WaveltempobasieRausc.pdf

[44] S. Eichner, G. Scheller, and U. Wessely, *Bewegungskompensierte Rauschreduktion von hochaufgelösten Filmsequenzen im Wavelet-Bereich*, ser. ITG-Fachbericht, R. Kays, Ed. VDE-Verlag, September 2005, vol. 188.

[45] FilmLight Ltd. Online: http://www.filmlight.ltd.uk

[46] General-purpose GPU. (2009). Online: http://www.gpgpu.org/

[47] R. Hartenstein, A. Hirschbiel, and M. Weber, "MOM - Map Oriented Machine," in *Proceedings of the International Workshop on Hardware Accelerators*, 1987.

[48] R. Heidenreich, "Design and implementation of a Video Interface Controller," Studienarbeit, IDA, TU-Braunschweig, August 2002, STA3334.

[49] S. Heithecker, "Communication and Memory Scheduling in Reconfigurable Image Processing Systems," Ph.D. dissertation, Technische Universität Braunschweig, 2008.

[50] S. Heithecker, A. do Carmo Lucas, and R. Ernst, "A Mixed QoS SDRAM Controller for FPGA-Based High-End Image Processing," in *Workshop on Signal Processing Systems Design and Implementation*. IEEE, 2003, p. TP.11.

[51] S. Heithecker, A. do Carmo Lucas, and R. Ernst, "FlexFilm - an Image Processor for Digital Film Processing," in *Dynamically Reconfigurable Architectures*, ser. Dagstuhl Seminar Proceedings, P. M. Athanas, J. Becker, G. Brebner, and J. Teich, Eds., no. 06141. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, April 2006, <http://drops.dagstuhl.de/opus/volltexte/2006/737> [date of citation: 2006-04-07]. Online: http://drops.dagstuhl.de/opus/volltexte/2006/737/

[52] S. Heithecker, A. do Carmo Lucas, and R. Ernst, "A High-End Real-Time Digital Film Processing Reconfigurable Platform," *EURASIP Journal on Embedded Systems, Special Issue on Dynamically Reconfigurable Architectures*, vol. 2007, pp. Article ID 85 318, 15 Pages, 2007.

[53] S. Heithecker and R. Ernst, "Traffic Shaping for an FPGA-Based SDRAM Controller with Complex QoS Requirements," in *Design Automation Conference (DAC)*. ACM, June 2005, pp. 575 – 578.

[54] J. L. Hennessy and D. A. Patterson, *Computer Architecture*, 3rd ed. Morgan Kaufmann, 2003.

[55] K. Henriss, P. Rüffer, and R. Ernst, "A Reconfigurable Hardware Platform for Digital Real-Time Signal Processing in Television Studios," in *Symposium on Field-Programmable Custom Computing Machines*. IEEE, April 2000.

[56] K. Henriss, P. Rüffer, and R. Ernst, "Arrangement for Processing Digital Video Signals in Realtime," International Patent # WO 01/80549 A1, October 2001. Online: http://v3.espacenet.com/textdoc?IDX=EP1417830

[57] HiTech Global, "Xilinx Virtex-5 LX330T, FX200T, SX240T PCI Express (Gen 1 & Gen 2), PPC 440, DSP, & RocketIO GTP/GTX Platform," Online, 2010, http://www.hitechglobal.com/Boards/PCIExpressLX330T.htm.

[58] Y.-W. Huang, C.-Y. Chen, C.-H. Tsai, C.-F. Shen, and L.-G. Chen, "Survey on Block Matching Motion Estimation Algorithms and Architectures with New Results," *Journal of VLSI Signal Processing*, vol. 42, pp. 297–320, 2006.

[59] A. Humber. (2010, October) NVIDIA Tesla GPUs Power World's Fastest Supercomputer. Press release. NVIDIA Corporation. Online: http://pressroom.nvidia.com/easyir/customrel.do?easyirid=A0D622CE9F579F09&prid=678988&releasejsp=release_157

[60] IBM. Online: http://www.ibm.com

[61] Impulse Accelerated Technologies. Online: http://www.impulsec.com

[62] InfiniBand Trade Association, *InfiniBand™ Architecture Specification Volume 1 Release 1.2*, October 2004.

[63] Institute of Computer and Network Engineering, Braunschweig University of Technology, Braunschweig. Online: http://www.ida.ing.tu-bs.de

[64] R. Iqbal, "Hardware bidirectional real time motion estimator on a Xilinx VirtexII Pro FPGA," Master's thesis, Institute of technology, Linköping University, December 2005, DPA3405.

[65] K. Jack, *Video Demystified*, 3rd ed. LLH Technology Publishing, 2001.

[66] J. Jacobs, W. Bond, R. Pouls, and G. J. Smit, "High Volume Colour Image Processing with Massively Parallel Embedded Processors," in *Parallel Computing: Current & Future Issues of High-End Computing*, ser. NIC, vol. 33. Julich: John von Neumann Institute for Computing, 2006, pp. 583–590. Online: http://doc.utwente.nl/57034/

[67] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell multiprocessor," *IBM Journal of Research and Development*, vol. 49, no. 4/5, pp. 589–604, July/September 2005.

[68] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM J. Res. Dev.*, vol. 49, pp. 589–604, July 2005. Online: http://portal.acm.org/citation.cfm?id=1148882.1148891

[69] R. Karp and R. Miller, "Properties of a Model for Parallel Computations: Determinancy, Termination, Queueing," *SIAM Journal of Applied Math*, vol. 40, no. 6, November 1966.

[70] B. Khailany, W. J. Dally, and S. Rixner, "Imagine: Media Processing with Streams," *IEEE Micro*, pp. 35–46, March/April 2001.

[71] J. Kong, M. Dimitrov, Y. Yang, J. Liyanage, L. Cao, J. Staples, M. Mantor, and H. Zhou, "Accelerating MATLAB Image Processing Toolbox functions on GPUs," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU '10. New York, NY, USA: ACM, 2010, pp. 75–85. Online: http://doi.acm.org/10.1145/1735688.1735703

[72] H. T. Kung and C. E. Leiserson, "Systolic Arrays (for VLSI)," in *Sparse matrix*, I. S. Duff and G. W. Stewart, Eds., vol. 8, Society for Industrial and Applied Mathematics. SIAM, 1978, pp. 256–282.

[73] M. Langhammer and M. Strickland, "FPGA Coprocessing Evolution: Sustained Performance Approaches Peak Performance," White Paper, June 2009.

[74] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," in *Proceedings of the IEEE*, vol. 75, no. 9, September 1987, pp. 1235 – 1245.

[75] P. Lieverse, E. Deprettere, A. C. J. Kienhuis, and E. de Kock, "A clustering approach to explore grain-sizes in the definition of weakly programmable processing elements," in *De Montfort University*, 1997, pp. 107–120.

[76] S. Maeda, S. Asano, T. Shimada, K. Awazu, and H. Tago, "A Real-Time Software Platform for the Cell Processor," *IEEE Micro*, vol. 25, pp. 20–29, 2005.

[77] G. M. Mark Priscaro, Jens Neuschaefer. (2010, October) Widespread Adoption of NVIDIA CUDA Accelerates Broadcast & Film Production. Press release. NVIDIA Corporation. International Broadcasting Convention (IBC) 2010. Online: http://pressroom.nvidia.com/easyir/customrel.do?easyirid=A0D622CE9F579F09&prid=660587&releasejsp=release_157

[78] S. Maya-Rueda, C. Torres-Huitzil, and M. Arias-Estrada, "A real-time FPGA-based architecture for optical flow computation," in *Proc. IEEE International Workshop on Computer Architectures for Machine Perception*, 12–16 May 2003, pp. 213–220.

[79] I. D. McCallum, "Intel® QuickAssist Technology FSB-FPGA Accelerator Architecture," in *Intel Developer FORUM*, 2007.

[80] (2008) Mercury. Online: http://www.mc.com/

[81] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. (2010, November) Top 500 supercomputer sites. TOP500.Org. Online: http://www.top500.org/

[82] M. Mitchell, "Building Signal-Processing Applications for the Cell Broadband Engine Processor," *Sourcery VSIPL++*, p. 15, July 2007. Online: www.codesourcery.com

[83] G. Moore, "Intel Drives Moore's Law Forward with 65 Nanometer Process Technology," *Press release*, p. 2, 2004.

[84] Nallatech Inc. Online: http://www.nallatech.com

[85] Nallatech Limited, "DIMEtalk V3.0 Application Development Environment," Product Brief, September 2006.

[86] Nallatech Limited, "BenONE-PCIe, 8-lane PCI Express FPGA motherboard," Product Brief, January 2010.

[87] S. Note, W. Geurts, F. Catthoor, and H. De Man, "Cathedral-iii: Architecture-driven high-level synthesis for high throughput dsp applications," in *Proceedings of the 28th ACM/IEEE Design Automation Conference*, ser. DAC '91. New York, NY, USA: ACM, 1991, pp. 597–602. Online: http://doi.acm.org/10.1145/127601.127739

[88] NVidia. Online: http://www.nvidia.com

[89] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*. Upper Saddle River, NJ, USA: Prentice Hall Press, 1989.

[90] Pandora. Online: http://pogle.pandora-int.com

[91] D. A. Patterson and J. L. Hennessy, *Computer architecture: a quantitative approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990.

[92] D. Pellerin, "Send in the Clown Fish:Implementing Video Analysis In Xilinx FPGAs," *Embedded magazine*, vol. Excellence in Embedded Magazine, p. 5, April 2009.

[93] A. Pohl, "Extension and improvement of a Local Memory Controller for an FPGA based videosystem," Studienarbeit, IDA, TU-Braunschweig, 2005, STA3344.

[94] R. Prince, "Colorfront team wins academy award®," Colorfront, Tech. Rep., January 2010.

[95] M. Priscaro. (2011, February) NVIDIA Quadro Powers All Five Academy Award Nominees for Best Visual Effects. Press release. NVIDIA Corporation.

[96] Quantel Ltd. Online: http://www.quantel.com

[97] D. Ramsey. (2009, March) UC San Diego and IBM Launch Center for Next-Generation Digital Media to Power Tomorrow's Virtual Worlds. University of California at San Diego. Online: http://ucsdnews.ucsd.edu/newsrel/general/03-09IBM.asp

[98] Red Digital Cinema. Online: http://www.red.com

[99] D. Q. Ren, "Algorithm level power efficiency optimization for CPU-GPU processing element in data intensive SIMD/SPMD computing," *J. Parallel Distrib. Comput.*, vol. 71, pp. 245–253, February 2011. Online: http://dx.doi.org/10.1016/j.jpdc.2010.10.007

[100] S. Riley, "How To Use FPOAs in Image Processing," *EETimes*, p. 4, 2007.

[101] S. Rixner, W. J. Dally, and U. J. Kapasi, "Memory Access Scheduling," in *International Symposium on Computer Architecture*, 2000, pp. 128–138.

[102] S. Rout, "Orthogonal vs. Biorthogonal Wavelets for Image Compression," Master's thesis, Virginia Polytechnic Institute and State University, 2003.

[103] H. Sahlbach, W. Putzke-Röming, S. Whitty, and R. Ernst, *Dynamic System Reconfiguration in Heterogeneous Platforms - The MORPHEUS Approach*. Springer Netherlands, 2009, vol. 40, ch. Real-Time Digital Film Processing, pp. 185–193.

[104] H. Sahlbach, S. Whitty, A. do Carmo Lucas, and R. Ernst, "FlexWAFE: FPGA-basierte Bildverarbeitung für digitales Kino," *FKT*, vol. 1-2/2010, p. 6, 2010.

[105] C. Sanz, M. J. Garrido, and J. M. Meneses, "VLSI Architecture for Motion Estimation using the Block-Matching Algorithm," in *EDTC*, 1996, p. 310.

[106] C. E. Shannon, "Communication in the Presence of Noise," in *IRE*, vol. 47, no. 1, January 1949, pp. 10–21. Online: http://www.stanford.edu/class/ee104/shannonpaper.pdf

[107] Silicon Graphics Inc. (SGI). Online: http://www.sgi.com

[108] Sony. Online: http://www.sony.com

[109] SRC computers. Online: http://www.srccomp.com/

[110] SRC Computers, Inc., "SRC Carte™ and Graphical User Interface (GUI) Programming," Whitepaper, January 2008.

[111] Storm-1. (2009) Stream Processors Inc. Online: http://www.streamprocessors.com/

[112] D. Thalmann, C. Hery, S. Lippman, H. Ono, S. Regelous, and D. Sutton, "Crowd and group animation," in *ACM SIGGRAPH 2004 Course Notes*, ser. SIGGRAPH '04. New York, NY, USA: ACM, 2004. Online: http://doi.acm.org/10.1145/1103900.1103934

[113] The MathWorks. Online: http://www.mathworks.com/

[114] Post-production products. Thomsom Grass Valley. Online: http://www.thomsongrassvalley.com/products_post/

[115] Thomson. (2008) Scream 4K/2K Resolution-Independent Grain Reducer. Thomson Grass Valley. Online: http://www.thomsongrassvalley.com

[116] Thomson Grass Valley. Online: http://www.thomsongrassvalley.com

[117] Bones. Thomson Grass Valley. Online: http://www.thomsongrassvalley.com/##bones##/

[118] Sprit4k. Thomson Grass Valley. Online: http://www.thomsongrassvalley.com/##spirit4k##/

[119] Viper filmstream. Thomson Grass Valley. Online: http://www.thomsongrassvalley.com/##viper##/

[120] Toshiba. Online: http://www.toshiba.com

[121] J. Tripp, M. Gokhale, and K. Peterson, "Trident: From high-level language to hardware circuitry," *Computer*, vol. 40, no. 3, pp. 28 –37, 2007.

[122] M. Wenzel, "Design and Implementation of a Hardware Motion Compensatior on a Xilinx Virtex II Pro FPGA," Diploma Thesis, IDA, TU-Braunschweig, April 2007.

[123] S. Whitty, H. Sahlbach, R. Ernst, and W. Putzke-Röming, "Mapping of a Film Grain Removal Algorithm to a Heterogeneous Reconfigurable Architecture," in *Proceedings of Design, Automation and Test in Europe (DATE)*, April 2009, pp. 27–32.

[124] D. Wick, S. Riley, and D. Lupia, "Radiation Hardened Field Programmable Object Array (FPOA) for Space Processing," in *MAFA*, 2007.

[125] N. Wilms, "Local Memory controller für ein FPGA basiertes Videosystem," Studienarbeit, IDA, TU-Braunschweig, October 2003.

[126] Xilinx Inc. Online: http://www.xilinx.com

[127] Xilinx Inc., "Celebrating 20 years of innovation," *Xilinx Xcell Journal*, vol. 48, p. 14–16, 2004.

[128] Xilinx Inc., "DS083: Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet," March 2007.

[129] N. Zervas, G. Anagnostopoulos, V. Spiliotopoulos, Y. Andreopoulos, and C. Goutis, "Evaluation of Design Alternatives for the 2D-Discrete Wavelet Transform," in *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 11, no. 12, December 2001, pp. 1246–1262.

[130] G. Zhou, "DWT Implementation in VHDL," Studienarbeit, IDA, TU-Braunschweig, November 2004, STA3337.